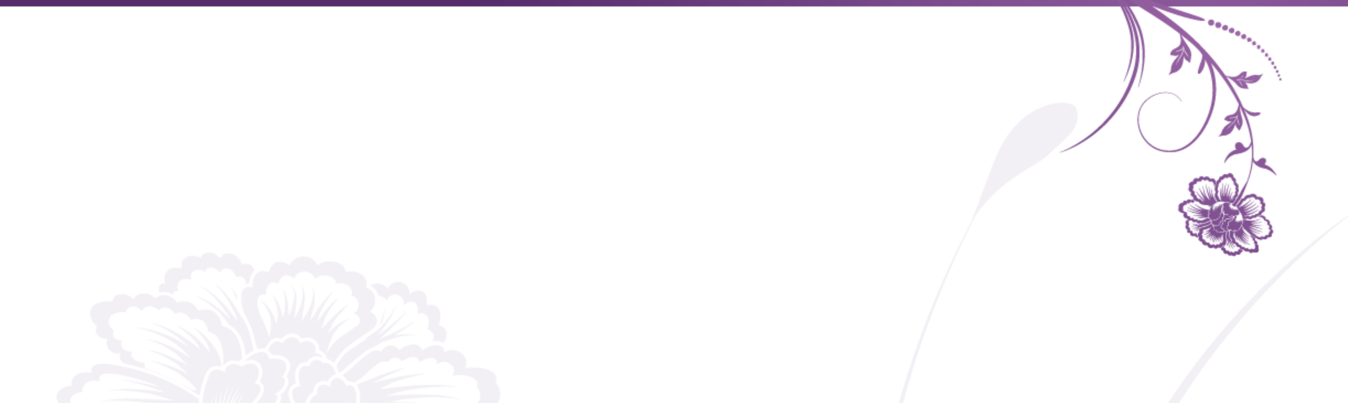
21世纪高等学校规划教材 | 计算机科学与技术





数据结构教程与题解

胡圣荣 周霭如 罗穗萍 编著

清华大学出版社

21世纪高等学校规划教材 | 计算机科学与技术



数据结构教程与题解

胡圣荣 周霭如 罗穗萍 编著

清华大学出版社
北京

内 容 简 介

本书介绍了线性表、栈、队列、串、多维数组、广义表、树、图、查找表、排序、文件等多种基本而常用的数据结构的数据表示和数据处理方法，包括逻辑结构、存储结构、基本运算及相应的算法，其中算法描述采用 C 语言。

本书力求通俗易懂，概念明确；部分课后练习和参考答案可作为正文的补充，如一些算法的实现、个别较深入的问题或证明推导等。

本书可作为计算机和信息类相关专业的本（专）科“数据结构”课程的教材和参考书。

图书在版编目（CIP）数据

数据结构教程与题解 / 胡圣荣，周霭如，罗穗萍编著. —北京：清华大学出版社，2011.9（2016.7 重印）
（21 世纪高等学校规划教材・计算机科学与技术）
ISBN 978-7-302-25664-9

Ⅰ. ①数…　Ⅱ. ①胡…　②周…　③罗…　Ⅲ. ①数据结构-高等学校-教材　Ⅳ. ①TP311.12

中国版本图书馆 CIP 数据核字（2011）第 164906 号

责任编辑： 高买花
责任校对： 时翠兰
责任印制： 何　芊

出版发行： 清华大学出版社
　　网　　址： http://www.tup.com.cn, http://www.wqbook.com
　　地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
　　社 总 机： 010-62770175　　**邮　　购：** 010-62786544
　　投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
　　质 量 反 馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者： 虎彩印艺股份有限公司
经　　销： 全国新华书店
开　　本： 185mm×260mm　　**印　张：** 21　　**字　　数：** 507 千字
版　　次： 2011 年 9 月第 1 版　　**印　　次：** 2016 年 7 月第 4 次印刷
印　　数： 4001～4500
定　　价： 33.00 元

产品编号：039972-01

编审委员会成员

（按地区排序）

扬州大学	李　云	教授
南京大学	骆　斌	教授
	黄　强	副教授
南京航空航天大学	黄志球	教授
	秦小麟	教授
南京理工大学	张功萱	教授
南京邮电学院	朱秀昌	教授
苏州大学	王宜怀	教授
	陈建明	副教授
江苏大学	鲍可进	教授
中国矿业大学	张　艳	教授
	姜　薇	副教授
武汉大学	何炎祥	教授
华中科技大学	刘乐善	教授
中南财经政法大学	刘腾红	教授
华中师范大学	叶俊民	教授
	郑世珏	教授
	陈　利	教授
江汉大学	颜　彬	教授
国防科技大学	赵克佳	教授
	邹北骥	教授
中南大学	刘卫国	教授
湖南大学	林亚平	教授
西安交通大学	沈钧毅	教授
	齐　勇	教授
长安大学	巨永锋	教授
哈尔滨工业大学	郭茂祖	教授
吉林大学	徐一平	教授
	毕　强	教授
山东大学	孟祥旭	教授
	郝兴伟	教授
中山大学	潘小轰	教授
厦门大学	冯少荣	教授
仰恩大学	张思民	教授
云南大学	刘惟一	教授
电子科技大学	刘乃琦	教授
	罗　蕾	教授
成都理工大学	蔡　淮	教授
	于　春	讲师
西南交通大学	曾华燊	教授

出版说明

随着我国改革开放的进一步深化，高等教育也得到了快速发展，各地高校紧密结合地方经济建设发展需要，科学运用市场调节机制，加大了使用信息科学等现代科学技术提升、改造传统学科专业的投入力度，通过教育改革合理调整和配置了教育资源，优化了传统学科专业，积极为地方经济建设输送人才，为我国经济社会的快速、健康和可持续发展以及高等教育自身的改革发展做出了巨大贡献。但是，高等教育质量还需要进一步提高以适应经济社会发展的需要，不少高校的专业设置和结构不尽合理，教师队伍整体素质亟待提高，人才培养模式、教学内容和方法需要进一步转变，学生的实践能力和创新精神亟待加强。

教育部一直十分重视高等教育质量工作。2007 年 1 月，教育部下发了《关于实施高等学校本科教学质量与教学改革工程的意见》，计划实施“高等学校本科教学质量与教学改革工程（简称‘质量工程’）”，通过专业结构调整、课程教材建设、实践教学改革、教学团队建设等多项内容，进一步深化高等学校教学改革，提高人才培养的能力和水平，更好地满足经济社会发展对高素质人才的需要。在贯彻和落实教育部“质量工程”的过程中，各地高校发挥师资力量强、办学经验丰富、教学资源充裕等优势，对其特色专业及特色课程（群）加以规划、整理和总结，更新教学内容、改革课程体系，建设了一大批内容新、体系新、方法新、手段新的特色课程。在此基础上，经教育部相关教学指导委员会专家的指导和建议，清华大学出版社在多个领域精选各高校的特色课程，分别规划出版系列教材，以配合“质量工程”的实施，满足各高校教学质量和教学改革的需要。

为了深入贯彻落实教育部《关于加强高等学校本科教学工作，提高教学质量的若干意见》精神，紧密配合教育部已经启动的“高等学校教学质量与教学改革工程精品课程建设工作”，在有关专家、教授的倡议和有关部门的大力支持下，我们组织并成立了“清华大学出版社教材编审委员会”（以下简称“编委会”），旨在配合教育部制定精品课程教材的出版规划，讨论并实施精品课程教材的编写与出版工作。“编委会”成员皆来自全国各类高等学校教学与科研第一线的骨干教师，其中许多教师为各校相关院、系主管教学的院长或系主任。

按照教育部的要求，“编委会”一致认为，精品课程的建设工作从开始就要坚持高标准、严要求，处于一个比较高的起点上；精品课程教材应该能够反映各高校教学改革与课程建设的需要，要有特色风格、有创新性（新体系、新内容、新手段、新思路，教材的内容体系有较高的科学创新、技术创新和理念创新的含量）、先进性（对原有的学科体系有实质性的改革和发展，顺应并符合 21 世纪教学发展的规律，代表并引领课程发展的趋势和方向）、示范性（教材所体现的课程体系具有较广泛的辐射性和示范性）和一定的前瞻性。教材由个人申报或各校推荐（通过所在高校的“编委会”成员推荐），经“编委会”认真评审，最后由清华大学出版社审定出版。

目前，针对计算机类和电子信息类相关专业成立了两个“编委会”，即“清华大学出版社计算机教材编审委员会”和“清华大学出版社电子信息教材编审委员会”。推出的特色精品教材包括：

（1）21 世纪高等学校规划教材·计算机应用——高等学校各类专业，特别是非计算机专业的计算机应用类教材。

（2）21 世纪高等学校规划教材·计算机科学与技术——高等学校计算机相关专业的教材。

（3）21 世纪高等学校规划教材·电子信息——高等学校电子信息相关专业的教材。

（4）21 世纪高等学校规划教材·软件工程——高等学校软件工程相关专业的教材。

（5）21 世纪高等学校规划教材·信息管理与信息系统。

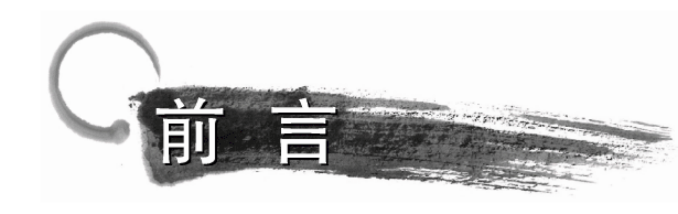
（6）21 世纪高等学校规划教材·财经管理与应用。

（7）21 世纪高等学校规划教材·电子商务。

（8）21 世纪高等学校规划教材·物联网。

清华大学出版社经过二十多年的努力，在教材尤其是计算机和电子信息类专业教材出版方面树立了权威品牌，为我国的高等教育事业做出了重要贡献。清华版教材形成了技术准确、内容严谨的独特风格，这种风格将延续并反映在特色精品教材的建设中。

清华大学出版社教材编审委员会
联系人：魏江江
E-mail:weijj@tup.tsinghua.edu.cn



前言

随着计算机软、硬件技术的迅速发展，计算机的应用越来越普及和广泛。但不管计算机作何用途，每一项应用总是某个程序的运行，用计算机解决任何问题都离不开程序设计。程序设计的实质就是数据的表示和数据的处理，数据结构就是研究这两个方面的一些基本问题的，包括如何有效地组织数据、数据元素之间是什么关系、数据在计算机中如何表示以及如何对数据进行有效的操作等。

“数据结构”是计算机和信息类专业的一门专业基础课程，特别对计算机软件和应用专业还是一门核心课程。在众多的计算机系统软件和应用软件中，都要用到各种数据结构。仅靠掌握几种计算机程序语言是难以应付众多复杂问题的，要想有效地使用计算机解决实际问题，必须积极主动地学习和应用数据结构的有关知识。数据结构对设计高性能的程序和软件至关重要。

数据结构的内容非常丰富，除了基本知识外，贯穿全课程的链表结构和递归技术，以及隐含在各算法中的技术和方法等都是学习中的重点和难点。多年的教学经验表明，选择一本合适的数据结构教材对这门课程的教学至关重要，过分抽象、过分简单，或者过分复杂，都不利于学生的学习，也不利于教师的讲授。鉴于此，本书的编写力求通俗易懂，概念明确，并对有些内容进行了取舍。本书的课后习题不仅仅是为了复习和巩固各章的知识，还常常安排成有关内容的补充、细化和深化，以适应有兴趣深入学习的读者；附录的参考答案中一般给出了较完整的解答。

在本书的编写中参考了众多相关教材和辅导材料，参考文献中未能一一列出，在此诚挚地对这些作者和编者表示衷心的感谢。

本书是在2003年出版的《数据结构教程与题解（用C/C++描述）》基础上，经过几年的不断修订而成，除了更新和改写了大量内容外，也补充了一些内容，如KMP算法、广义表的储存结构、外排序简介，以及通过脚注、附录的形式指出了文献中一些含义有别的名称，一些结论、公式、定理等的推导等，使全书更加完整。另外，在附录中还给出了几个与上机实验、编程有关的问题。

虽然编者投入了大量时间和精力，但限于水平和能力，书中仍难免存在不足和错误之处，恳请各位专家、读者批评指正。

编　者

2011.3

编者邮箱：hsrzz@21cn.com

目录

第1章 概论

随着计算机的普及和软硬件技术的发展，计算机的应用越来越广泛，但不管计算机作何用途，每一项应用总是某个程序的运行。所以，用计算机解决任何问题都离不开程序设计，而程序设计的实质就是数据的表示和数据的处理，数据结构就是研究这两个方面的一些基本问题的，包括如何组织数据、数据元素之间是什么关系、数据在计算机中如何表示以及如何对数据进行操作等。数据结构对设计高性能程序和软件至关重要。

本章介绍了数据结构的基本概念，包括数据的逻辑结构、存储结构、基本运算和运算的实现以及算法分析等。

1.1 引言

在现实生活中，当我们谈到事物的“结构”时，一般是指它由哪些部分组成，各部分之间的相互关系如何等，如对于计算机课程的体系结构，我们会关心它有哪些课程、各课程之间的关系如何等。所以，对“数据结构”这个概念，从字面上可以理解为数据的组成和相互间的关系，或称数据的组织形式。不过这并不全面，因为数据结构中还应包含数据的相关操作。本章后面会逐步对数据结构进行解释，其中会涉及很多相关概念。这里先看另一个问题：我们为什么要学习数据结构？或者说学习数据结构有什么用？

简单地说，学习数据结构是我们编程的需要。这是因为，不论什么程序，它本质上都是计算机对某种数据的加工处理，计算机相当于一个处理机，数据是它加工处理的“原料”。这里涉及两个基本问题：首先，数据要存储到计算机中，才能被计算机加工处理；其次，如何对数据进行处理。前一个问题称为**数据表示**，后一个问题称为**数据处理**。所以，程序设计的实质就是数据的表示和数据的处理。

数据在计算机存储器内的存在形式称为机内表示，这之前的数据表现形式称为机外表示，所以数据表示的工作就是将数据从机外表示转化为机内表示。在数据表示中，不是简单地将数据的值存储到计算机中就可以了，还要直接或间接地存储数据之间的相互关系，对关系的表示常常是一些复杂问题的关键。数据处理的工作就是用计算机可执行语句编制程序，描述对已存入计算机的数据进行各种具体操作，继而完成整个处理任务。数据结构就是研究程序设计过程中数据表示和数据处理这两个方面的一些基本问题的。

或许有人会问：我们以前学习高级语言如C/C++语言时，并没有学过数据结构，不是一样也编出了很多程序并且运行得很好吗？这里有一个认识问题。首先，我们在学习高级

语言时所编制的程序基本上是属于数值计算型的，如级数求和、方程求根等，所涉及的运算对象比较简单，如整数、实数等，数据结构的问题不明显；其次，即便如此，我们在编程中也不知不觉地、或多或少地使用了数据结构的一些知识或方法。最后，从数据结构的观点看，即使是一个单独的数据，如一个整数或字符，也可看成一个简单的数据结构。

下面先看一个简单的例子。

例 1.1 方程求根：

$f(x)=a_2x^2+a_1x+a_0=0$

解：首先要把方程的系数存入计算机，如用 3 个变量 a_0、a_1、a_2 表示，然后才能进行具体的求根计算。这就是我们在学习高级语言时一般采用的方法。但对一般形式的方程：

$f(x)=a_nx^n+a_{n-1}x^{n-1}+\cdots+a_1x+a_0=0$

我们一般就不会对每个系数都用单独的变量名来表示了，因为变量多，对名字的管理和书写都很不方便，如 20 次的多项式，就需要 21 个变量名。这时，我们一般会用一个数组如 A[21]来集中存放这些系数，这样只需记住一个名字，即数组名，然后通过数组下标来区分各个系数，如 A[i]表示 a_i。这里，数组就是数据结构中一种比较简单的存储结构——顺序存储结构，它将各个元素在存储空间上连续存放。

如果多项式的次数不定，上述方法又有问题，如 C/C++等语言中数组的大小一经定义后就不能再变化，这时，如果数组定义得较大，而多项式的次数低，实际系数少，数组空间就有浪费现象；反之，如果数组定义得较小，而多项式的次数高，实际系数多，数组空间就可能不足而产生溢出。为适应数据个数变化的情况，我们可用内存分配函数为每个系数动态分配空间，并通过指针将它们联系起来，即得到链表。这里，链表也是数据结构中常用的一种存储结构——链式存储结构。

如果再考虑到多项式的系数中通常有很多为零，为了节省存储空间，并和书写习惯一致，我们可不存储零系数，但这时要把非零系数和原多项式之间的关系表示清楚，就涉及数据的压缩存储问题，也是数据结构里要研究的内容。

在上面存放各系数时，不论数组还是链表，我们显然不会胡乱存储，而是很自然地按照系数间的“前后”次序，即按对应项的次数从高到低或从低到高来进行。这实际上说明了这样一个问题：这些系数间是有相互关系的，对本问题，它们按次数的高低顺序形成一个有穷序列(a_0, a_1, a_2, …, a_n)，对任一个系数 a_i，排在它前面的与之相邻的系数以及排在它后面的与之相邻的系数都最多只有一个，这是数据结构里一种比较简单的逻辑结构——线性结构。

将系数存储到计算机中后，就可以进行求根计算了。对低次方程如 2 次、3 次，可用求根公式，但一般情况下要采用迭代法，如牛顿迭代法。在计算中，要涉及 x 的各次方 x^i，显然不必每个都单独计算，否则会有很多重复计算，如 x^3 不必用 x*x*x 来计算，可以在已有 x^2 的基础上再做一次乘法 $x*x^2$ 即可。这里，怎样实现具体的计算方法，属于算法问题，而怎样评价算法的效率，就是算法分析问题。这都是数据结构里要讨论的内容。

可见，对这个例题，我们已经“无意间”用到了数据结构的一些相关知识，同时也看到，即使问题不太复杂，如果只有程序语言方面的知识，将数据如何有效地组织起来，以及如何对数据进行有效的运算已显得有些“力不从心”了。

上述例子属于数值计算问题，在计算机发展初期，人们使用计算机主要就是处理这类

问题的。由于所涉及的数据对象比较简单，如整型、实型或布尔型等，数据结构的问题不突出，程序设计者的主要精力集中在程序设计的技巧上，解决此类问题的关键是数值计算方法（算法），程序以算法为中心。

但是，大量的实际问题仅凭高级语言的知识是无法处理的，必须主动地借助于数据结构的知识。这是因为，随着计算机软硬件的发展和计算机的普及，计算机应用领域不断扩大，早已不再局限于科学计算了，大量的应用，如文字处理、数据库、多媒体、游戏、过程控制等都是非数值型问题。在这些问题中，要处理的数据一般比较复杂，如字符、表格、图像、声音等，这不仅体现在数据的内容比较复杂，更主要的是数据之间还有复杂的关系，而这些关系无法用数学方程式描述；另外，对数据的处理一般也很复杂。于是，如何有效地组织数据以及如何对数据进行有效的运算等问题就成了处理这类问题的关键，而这正是数据结构所要研究的。

例 1.2　用计算机进行部门机构（如图 1.1 所示）管理。

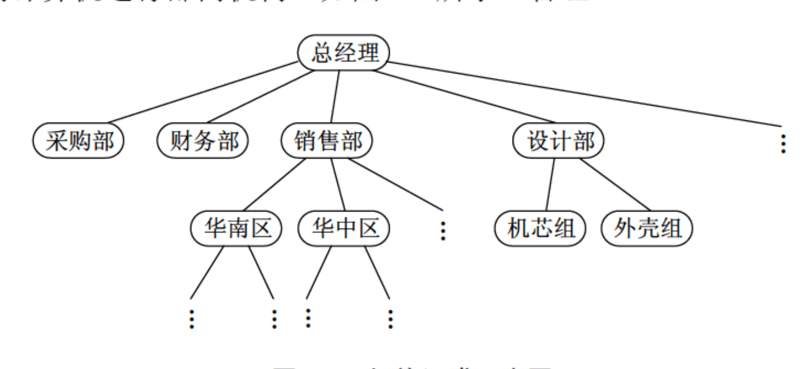


图 1.1　机构组成示意图

解：这里要处理的数据是整个机构，它由各个部门组成。显然要将各个部门的信息存储到计算机中，如员工数、位置、编号、名称等，我们可用结构（或称记录）类型的量来表示这些信息。然而，理顺部门之间的关系，或者说对关系的表示才是问题的关键：各部门间可能是上下级关系，也可能是同级关系，整个部门机构组成一种层次结构。如果不把这些关系表示出来，这个问题根本就没法解决。层次关系可很自然地用图 1.1 的形式表示，其中，每个部门用一个椭圆框表示，椭圆框间的连线表示上下级关系：每个部门最多只有一个直接上级，但可有多个直接下级。数据的这种层次结构在数据结构里称为树。

所以，在存储部门数据时，还要将部门间的关系存储起来或反映出来。我们可以将所有部门按某种次序依次存放到一片连续的内存单元中，并在每个部门中增加上下级关系信息，比如增加上级关系信息，这可用一个指针来表示，即指出它上级部门的存储位置。这样，从任一个部门通过指向上级的指针可找到它的所有上级；如果某个部门的上级指针为空，则它是最上级的部门；如果要考察某两个部门之间有没有上下级关系，则只要看其中某个部门的上级中有没有另一个就可以了。这是树形结构的一种双亲存储结构。

将部门数据存储到计算机中以后，就可对其进行有关的管理工作了，如新部门的增加（插入）、旧部门的撤销（删除）、部门的查询等，这些操作的具体实现过程就是本问题的算法。

上面两个例子用到的数据结构是线性表和树。一般说来，为了有效地表示数据，特别是数据之间的关系，需要设计和采用合适的数据结构（以将数据组织起来，不妨称此工作

为数据的“**结构化**”)，在此基础上才能写出有效的算法完成数据的处理，从而有效地解决实际问题。在实践中经常会遇到这种现象：同一个问题，采用一个“好”的数据结构很快就能完成运算，而采用一个“差”的数据结构，运算时间可能会相差几倍甚至几十倍或更多（比如本书后面将介绍的查找表，对不同的数据组织结构，相应查找算法的效率就会有巨大的差异)。

著名的瑞士计算机科学家沃思（N.Wirth）教授曾提出：**算法+数据结构=程序**。这里的数据结构是指数据的逻辑结构和存储结构，而算法则是对数据运算的描述。由此可见，程序设计的实质也可看成是对实际问题选择一种好的数据结构，加之设计一个好的算法，而好的算法在很大程度上取决于描述实际问题的数据结构。

在众多的计算机系统软件和应用软件中，都要用到各种数据结构。仅靠掌握几种计算机语言是难以应付众多复杂问题的，要想有效地使用计算机解决实际问题，必须积极主动地学习和应用数据结构的有关知识。

数据结构作为一门独立课程，是在 1968 年开始设立的，美国唐·欧·克努特教授开创了其最初体系。现在它是一门研究数据组织、存储和运算的一般方法的学科，介于数学、计算机硬件和计算机软件三者之间，是计算机软件和计算机应用专业的一门核心课程，也是一些计算机软件相关专业的重要课程。它不仅是一般程序设计的基础，也是设计和实现编译程序、操作系统、数据库系统及其他系统程序和大型应用程序的重要基础。

1.2 数据结构的概念

上节已涉及了数据结构的一些概念，但没有具体解释，下面对数据结构的一些概念和术语进行比较详细的介绍。

1.2.1 数据

从数据结构的观点看，通常所说的“数据”应分成三个不同的层次，即数据、数据元素和数据项。

数据（Data)：凡能被计算机存储、加工处理的对象通称为数据。它是计算机程序加工处理的对象和原料。前已指出，早期的计算机主要用于科学计算，数据的概念主要是指整型、实型或布尔型等数值型数据；随着计算机软硬件的发展和计算机的普及，计算机应用领域不断扩大，数据的概念已逐步扩展到字符串、表格、图像甚至语言等。

这里顺便提一下与数据密切相关的另一个概念：信息。简单地说，信息是加工处理后的数据，是数据的内涵；而数据是信息的载体，信息需要通过数据表示出来。如 A、B 两个人成绩分别是 85 和 95 分，我们得到的信息是他们的成绩都较好，但 B 更好。反之，为了说明一个人的成绩好，我们需要给出具体数据，如 90 分（或与之相关的量)。但有时数据和信息并不严格区分，如数据处理也称信息处理。

数据元素（Data Element)：数据的基本单位，在程序中作为一个整体加以考虑和处理，通常具有完整确定的实际意义。有些情况下，数据元素也称为元素、结点、顶点或记录。

数据项（Data Item）：数据不可分割的最小标识单位，具有独立含义，但通常不具有完整确定的实际意义，或不被当做一个整体看待。有时数据项也称为字段或域。数据元素一般由若干个数据项组成，但有时也可只含有一个数据项。

数据、数据元素和数据项反映了数据组织的三个层次：数据可由若干数据元素组成，数据元素又可由若干数据项组成。例如，对例 1.2 的部门机构组织问题，数据即指所有部门构成的整体；其中每个部门就是一个数据元素，因为在此问题中它被当做运算的基本单位。如删除、插入等运算作用的对象就是某个部门，不是整个机构，也不是某个部门中的个别项目。部门的名称、编号等项目则为数据项，它们只表示部门某一方面的信息，在本问题中单独存在时没有完整确定的实际意义。

1.2.2　数据类型

数据类型是与数据结构密切相关的一个概念。它最早出现在高级程序设计语言中，用以刻画程序中操作对象的特性。在用高级语言编写的程序中，每个变量、常量或表达式都有一个确定的数据类型。

数据类型（Data Type）是具有相同性质的计算机数据的集合及在这个数据集合上的一组操作的总称，它显式或隐式地规定了数据的取值范围和操作特性。例如，C/C++语言中的无符号字符型（unsigned char）代表闭区间[0, 255]中的整数，在这个整数集中可以进行加、减、乘、整除、取模等操作。

数据类型可以分为**原子类型**和**结构类型**（或称导出类型、复合类型）。原子类型的值是不可分解的，它由计算机语言提供，如 C/C++语言中的整型、字符型等；结构类型的值是可分解的，即由若干成分组成，并且这些成分本身还可以是结构的。结构类型要借用计算机语言提供的数据组织机制，由用户自己定义，如 C/C++语言中的结构、数组等。

抽象数据类型（Abstract Data Type，ADT）是指一个数学模型以及定义在该模型上的一组操作的总称。“抽象”的含义是指其逻辑特征与具体的软硬件实现（即计算机内部的表示和实现）无关。在用户看来，无论怎样实现，只要其数学特征不变，就不影响其外部使用。

抽象数据类型和数据类型实质上是一个概念。例如，各种计算机都拥有的整数类型就是一个抽象数据类型，在用户看来其数学特征相同，而实际上它们在不同处理器上的实现是可以不同的。但另一方面，抽象数据类型的范畴更广，它不局限于在各种处理器中已定义并实现的数据类型，还包括用户自己定义的数据类型。

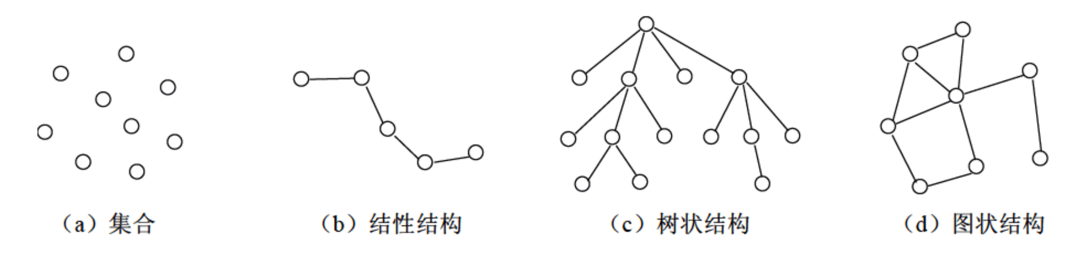
在定义抽象数据类型时，将一组数据和施加于这些数据上的一组操作封装在一起，用户程序只能通过在 ADT 里定义的某些操作来访问其中的数据，从而实现了信息的隐藏。在这个过程中，数据的表示及其操作的细节在模块的内部给出，在模块的外部使用的只是独立于具体实现的抽象的数据及抽象的操作。所以，抽象数据类型的特征是使用与实现相分离，实行封装和信息隐藏。

1.2.3　逻辑结构

为了表示数据间的关系，需要引入逻辑结构的概念。

数据元素对应着客观世界中的实体，数据元素之间必然存在着各种各样的关系，这种数据元素之间的关系就称为**结构**。其中，数据元素之间的关联方式（或称邻接关系）称做数据的**逻辑关系**，数据元素之间逻辑关系的整体称为**逻辑结构**（Logical Structure）。

为了讨论方便，数据的逻辑结构一般可用示意图表示。具体方法为，用小圆圈代表数据元素，用小圆圈之间的连线代表数据元素间的关系，如果强调关系的方向性，可用带箭头的线段表示关系。有四类基本的逻辑结构，如图 1.2 所示。



（a）集合　（b）结性结构　（c）树状结构　（d）图状结构

图 1.2　四种基本逻辑结构示意图

（1）**集合**：任何两点之间不考虑邻接关系或没有邻接关系，或称做没有关系的关系，其数据组织形式松散，元素之间是“平等”的，它们的共同关系是“属于同一个集合”。也可以说，集合中各元素之间除了“同属于一个集合”的关系外，别无其他关系。

（2）**线性结构**：有且仅有一个开始结点和一个终端结点，并且任何结点都最多只有一个直接前趋和一个直接后继。某点的**直接前趋**（Immediate Predecessor）是指与之相邻且在它前面的结点，**直接后继**（Immediate Successor）是指与之相邻且在其后的结点。开始结点没有前趋，终端结点没有后继。线性结构中数据元素之间存在一个对一个的关系。

（3）**树状结构**：除一个特殊元素（根）外，每个元素都只有一个直接前趋，但可有多个直接后继，结点之间具有分支、层次特性。树状结构中数据元素之间存在一个对多个的关系。

（4）**图状结构**：任何两点之间都可能邻接，结点之间形成网状结构。任一元素都可有多个直接前趋和多个直接后继，元素之间存在多个对多个的关系。

线性结构是一种最常见的数据结构，本书第 2 章、第 3 章介绍的线性表、栈、队列、串等均为线性结构。树状结构与图状结构也称为**非线性结构**，它的逻辑特征是一个结点可能有多个直接前趋或多个直接后继。集合比较特殊，可把它归到非线性结构，因为“线性”之外都是“非线性”，但在实际使用时，也经常对其数据元素增加某种“线性”关系，如出现的先后次序等，按线性结构处理（当然也可根据需要施加某种“非线性”关系，如分支、层次关系等，按相应的非线性结构处理）。

有一些数据结构，如多维数组和广义表，尽管本质上属于图状结构，但由于自身的具体特点，与图状结构的处理方法有很大不同，所以一般单独讨论。

关于逻辑结构，有以下几点需要特别注意：

（1）逻辑结构与数据本身的形式、内容无关。如机构组成、族谱管理等问题涉及的数据，其形式、内容完全不同，但都是树状结构；而同一问题，改变某个元素的名称或内容，逻辑结构并不受影响。

（2）逻辑结构与数据元素的相对位置无关。如机构组成中，进行部门关系重组，这时



结点间的相对位置可能改变，但整体上仍是树状结构。

（3）逻辑结构与所含结点的个数无关。如不同的机构组成，树状结构中结点数不同；而同一机构组成，增加或删除几个部门，其结果仍是树状结构。

由此可见，一些表面上很不相同的数据可以有相同的逻辑结构，因此，逻辑结构是数据组织的某种“本质性”的东西。事实上，逻辑结构是数据组织的主要方面。

在不至于混淆的情况下，本书以后常常将直接前趋简称为前趋、直接后继简称为后继。

1.2.4　存储结构

为了让计算机对数据进行处理，我们还要研究如何在计算机中表示数据。数据的存储实现（机内表示）称为数据的**存储结构**（Storage Structure）或物理结构，它是指数据元素及其关系在计算机存储器内的表示。一个存储结构一般应包括 3 个内容：

（1）**内容存储**。存储各数据元素的内容（值），每个数据元素占据独立的可访问的存储区。

（2）**关系存储**。直接或间接地（显式或隐式地）存储各数据元素间的逻辑关系。

（3）**附加存储**。一般是为便于运算实现而设置的辅助结点。

其中，前两个部分是所有存储结构都必须具备的，第三个部分则根据实际需要决定是否设置。由于数据元素内部的组织形式一般比较简单，内容存储也就比较简单，所以，逻辑关系的表示是存储结构的主要内容。数据元素存储后，元素间的逻辑关系便由存储结点间的关联方式间接表示。

这里要指出，数据的存储结构是逻辑结构用计算机语言的实现，它依赖于计算机语言和物理设备。但对计算机语言来说，存储结构是具体的，所以我们一般不用具体的物理存储地址来描述存储结构，而只在高级语言的层次上讨论存储结构。数据的存储结构可用以下 4 种基本的存储方式得到：

（1）**顺序存储方式**。所有结点相继存放到一片连续的存储区中，元素之间的逻辑关系通过物理位置关系间接表示。由此得到的存储表示称为**顺序存储结构**（Sequential Storage Structure），它是一种最基本的存储方法，通常借助程序设计语言中的数组来描述。该方式主要用于线性结构，非线性结构需要通过某种线性化的方法来实现。

（2）**链接存储方式**。结点之间的物理位置不一定连续，它们之间的逻辑关系通过附加的指针来表示。即每个元素的存储区分两大部分：一部分为数据区，存储元素本身的数据内容；另一部分为指针区，存储与其他元素之间的关系信息（一般为相关的其他元素的地址）。由此得到的存储表示称为**链式存储结构**（Linked Storage Structure），它通常借助于程序设计语言中的指针来描述，适合存储复杂的数据结构。

（3）**索引存储方式**。在存储结点信息的同时，还建立附加的索引表。索引表中的每一项称为索引项，索引项的一般形式是：（关键字，地址），其中关键字是能标识一个结点的那些数据项。若每个结点在索引表中都有一个索引项，则该索引表称为**稠密索引**（Dense Index），此时索引项地址指出该结点所在的存储位置；若一组结点在索引表中只对应一个索引项，则该索引表称为**稀疏索引**（Sparse Index），此时索引项的地址指示一组结点的起始存储位置。索引存储并不强调关系的存储，而是针对数据内容的，主要面向检索（查找）

操作。

（4）**散列存储方式**。以结点的关键字值为自变量，通过某个函数计算出该结点的存储位置（或位置区间端点）。这个函数称为散列函数。散列存储也是面向内容存储的。

实际上，在这 4 种存储方式中，顺序存储方式和链接存储方式是最基本的，因为索引存储方式和散列存储方式在具体实现时需要利用前两种结构，也可看成前两种结构的衍生。

上述四种存储方法，既可以单独使用，也可以组合使用。有时同一种逻辑结构可采用不同的存储结构，如何选择要视具体要求而定，主要是考虑运算的方便性及算法的时空要求。

1.2.5 运算

为了完成数据处理任务，需要引入运算的概念。

一般地，**运算**是指在逻辑结构上施加的操作，即对逻辑结构的加工。运算与逻辑结构紧密相连，每种逻辑结构都有一个运算的集合。运算的种类很多，随不同的应用而不同。根据操作的结果，运算可分为两种类型：

（1）加工型运算，其操作改变了原逻辑结构的“值”，如结点个数、结点内容等。

（2）引用型运算，其操作不改变原逻辑结构，只从中提取某些信息。

例如，在例 1.2 的树状结构 S 上，一般可定义以下运算：

- 查找。引用型运算，在 S 中寻找满足一定条件的结点。
- 插入。加工型运算，在 S 的指定位置上添加新的结点。
- 删除。加工型运算，删去 S 的某个指定结点。
- 读取。引用型运算，读取 S 中某指定位置上结点的内容。
- 更新[①]。加工型运算，更新 S 中某指定结点的内容。
- 遍历。引用型运算，按某种方式访问 S 中各结点，使得每个结点恰好被访问一次。
- 关系访问。引用型运算，访问 S 中特定关系的结点，比如结点的上下级、同级结点等。

根据实际需要，可对这些运算进行增减。

在各种运算中，如果某些运算，它的实现不能利用其他运算，而其他运算可以或需要利用该运算，则这些运算称为**基本运算**。如上面的更新运算就不是基本运算，因为在更新时，可先利用查找运算，找到该该点后再更改有关内容（或删除原结点，再插入新结点）；而查找运算是基本运算，它不能利用其他运算。每种逻辑结构都有自己的基本运算集，逻辑结构不同，基本运算集一般也不同。

一般地，我们将较复杂的运算分解为若干较简单的运算，有利于降低程序设计的难度，同时也有利于提高程序设计的效率。在简单运算中，再分解出一些基本运算，当基本运算实现后，其他运算就可通过调用基本运算来实现，进而完成整个程序。

1.2.6 算法

数据的运算是定义在逻辑结构上的，只指出“做什么”，而不考虑“怎么做”。运算实

---

① 有的文献用“修改”。但更新或修改本身并不专指结点内容的改动，只要有变动都可，如更新或修改数据库，可能是对其记录的插入、删除以及内容改动等。本书中“更新”主要指结点内容的更改，修改则可指各种情况的改动（具体看上下文）。

现的这个细节问题就是算法。算法是与数据结构密切相关的一个概念，讨论某种数据结构必然会涉及相应的算法，而设计某个算法也必然要涉及具体的数据结构。

所谓**算法**（Algorithm），通俗地讲，就是解决特定问题的方法和步骤；较严格地说，就是规则的有穷集合，这些规则规定了一个指令的有限序列，其中每条指令表示一个或多个操作。一个算法必须满足下述准则：

（1）输入。具有零个或多个输入，它们是算法开始前的初始量。

（2）输出。至少产生一个输出，它们是与输入有某种关系的量，是算法的执行结果。

（3）有穷性。算法的执行步骤（或每条指令的执行次数）必须是有限的，整个算法必须在有限步（或有限条指令）后结束。

（4）确定性。算法每一步（或每条指令）的含义都必须明确，无二义性。

（5）可行性[①]。算法每一步（或每条指令）是可执行的，并且执行时间是有限的，整个算法必须在有限时间内完成[②]。

这里要注意，本课程所指的算法是针对数据结构的，而一般问题的算法是面向应用的，它涉及数据结构的应用，但范围比数据结构中的运算要广。

另外，算法与程序的含义很相似，但二者是有区别的：

（1）程序不一定满足有穷性，即不一定是算法。如操作系统就不是一个算法，因为只要不遭破坏，它就永远不会停止，即使没有作业要处理，它仍处于等待循环中（不过操作系统内部每个具体任务的实现都应是一个算法)。一个程序如果对任何输入都不会陷入无限循环，就是一个算法。

（2）程序中的指令必须是机器可执行的，而算法中的指令虽要求可执行，但不一定是“机器可执行”。如果一个算法用机器可执行的语言来书写，则它就是一个程序，即该算法在计算机上的特定实现。

任何算法都必须用某种方法描述出来，即将算法中的**操作**及其**执行顺序**（简称算法两要素）描述出来，其中常用的就是用**语言描述**。根据描述语言的不同，一般可将算法分为以下三类：

（1）运行终止的程序可执行部分[③]。采用计算机程序设计语言描述，可直接在计算机上运行，从而使给定问题在有限时间内被机械地求解。这类算法比较严谨，但要熟悉计算机语言，有一定难度，也不太直观，常常需要通过注释来提高可读性。

（2）伪语言算法。采用伪程序设计语言描述，不能直接在计算机上运行。伪语言介于程序设计语言和自然语言之间，它忽略程序设计语言中一些严格的语法规则和细节描述，因此伪语言描述可突出算法设计的主要方面而不是语法细节，又比自然语言更接近程序。伪语言算法一般比较简洁，便于编写和阅读，适合于教学和交流，同时也容易修改成程序。

（3）非形式算法。采用自然语言，同时还可夹杂使用程序设计语言或伪程序设计语言（如流程控制语句 while、for、if 等）描述。这类算法简单易懂，但不够严谨。

---

① 有的文献用“有效性”。显然不可执行的指令是无效的。

② 有的文献把执行时间的有限性归入到“有穷性”。另外，时间的有限性应该在合理的范围内，如需耗时百年的计算一般不会认为是可行的。

③ 这里不写“程序语言算法”是指去除其中的非执行部分，如变量和函数的说明语句等。

算法除了用语言描述外，实际上还有一种**图形描述**方法，如流程图、N-S 图等，这种描述更加简单明了。但我们只把它看做语言描述的一个辅助手段，并且它最终还是要用语言描述出来。

不管算法用什么方法描述，它最终都要转换为程序才能在计算机上运行。上述几种描述方法的可读性依次增强，但可读性越强，离最终程序的距离越远。对一些较复杂问题，一次性地写出它的程序比较困难，一般是先写出伪语言算法或非形式算法，再通过“逐步求精”的过程转化为实际程序。逐步求精的过程也符合人们认识事物的思维活动。

原则上说，任何算法都可以用任一种程序设计语言来实现，但显然具体实现的难易程度和效果会有所不同。随着面向对象程序设计语言的流行，数据结构中越来越多地出现了这类语言的描述，如 C++描述等。应该说，为了较好地描述数据结构，特别是抽象数据类型，以及较好地解决代码重用等问题，这样做是有利的。但这同时也对读者提出了更高的要求，读者必须较好地掌握了面向对象的程序设计知识，否则可能出现数据结构的内容不突出，面向对象的内容倒成了问题的重点和难点。另外，类的构造、析构、继承和派生、重载、多态、模板等面向对象的概念和方法引入后，数据结构的内容常常显得复杂和“高深”起来，在使用中很多初学者都感到比较困难。

为了突出数据结构本身的内容而又不过于强调语言的细节，有些教材采用了伪语言，如类 C 等。但伪语言算法毕竟还要转换成程序语言算法，其中除了语法上的不严格外，伪语句和实际语句上的差别也经常引起程序问题，如调用程序和被调用程序间的信息是双向传递还是单向传递等。

本书对数据结构的描述采用了 C/C++语言，即主体部分为 C 语言，但对输入输出、宏常量、注释、动态内存分配和释放等采用了比较简洁的 C++扩展形式，如表 1.1 所示。

表 1.1　C/C++语句对比

C 语句	C++语句
`#include <stdio.h>`	`#include <iostream.h>`
…	…
`scanf("%d%c",&i,&ch);`	`cin>>i>>ch;`
`printf("%d%c\n",i,ch);`	`cout<<i<<ch<<endl;`
`#define maxsize 100`	`const int maxsize=100;`
`for(i=0;i<n;i++) s=s+a[i];/*元素求和*/`	`for(i=0;i<n;i++) s=s+a[i];//数组元素求和`
`int *p,*q;`	`int *p,*q;`
`p=malloc(10*sizeof(int));`	`p=new int[10]; //动态分配 10 个整数空间(数组)`
`q=malloc(sizeof(int));`	`q=new int;    //动态分配 1 个整数空间`
…	…
`free(p);`	`delete []p;    //释放数组空间`
`free(q);`	`delete q;`

C++还新增了一个功能——引用，它在参数传递上很方便，由于没有对应的 C 语句，本书暂未采用，但作为附录（附录 B）供参考。

由于采用了 C 的扩展部分，所以上机时需采用 C++编译器（如 Turbo C++ 3.0 等）。有关这些扩展的详细知识，请参考 C/C++语言的有关资料。

至于 C++的主要扩展——类，本书则未采用（原因前已叙及）。实践表明，掌握了数据结构的基本知识和方法后，可以非常方便地应用到 C++等面向对象的程序设计中。

1.2.7　数据结构

前面介绍了数据结构相关的几个概念，但一直没有说究竟什么是数据结构。事实上，对数据结构这一概念，目前也没有一致公认的定义。比较流行的观点有两种。一种观点认为，一个数据结构是由一个逻辑结构 S、一个定义在 S 上的基本运算集 Δ 和 S 的一个存储实现 D 所构成的整体(S, Δ, D)；另一种观点认为，一个数据结构是由一个逻辑结构 S 和定义在 S 上的一个基本运算集 Δ 构成的整体(S, Δ)。本书采用了前一种观点，将数据的逻辑结构、数据的存储结构及数据的运算这三方面看成一个有机的整体，这样，数据结构的定义为：

数据结构（Data Structure）是指相互间存在着一种或多种关系的数据元素的集合，它们按照某种逻辑关系组织起来，并用计算机语言，按一定的存储方式存储在计算机的存储器中，同时在这些数据上定义了一个运算的集合。简单地说，一个数据结构就是一类数据的表示及其相关操作，它一般包括三个方面的内容：数据的逻辑结构、数据的存储结构、数据的运算。

在不产生混淆的情况下，也常将数据的逻辑结构简称为数据结构。

数据的逻辑结构是数据本身所固有的，与计算机无关。每种逻辑结构都有自己的一组基本运算，它规定了数据的基本操作方式。这里的数据是指具体问题中要处理的对象，运算也来源于具体问题的处理要求。由一种逻辑结构和一组基本运算构成的整体，与数据的存储无关，也是独立于计算机的，可看成是从具体实际问题抽象出来的数学模型。将数据以及数据间的逻辑关系存储起来后，就得到了存储结构。数据的运算定义在逻辑结构上，只指出“做什么”，而不考虑“怎么做”，当确定了存储结构后，才能考虑如何将运算具体实现，即“怎么做”，这就是算法问题。由于存储结构是逻辑结构的实现，算法是运算的实现，所以，也可以认为数据结构的基本任务就是数据结构的设计和实现。

前面已指出，程序设计的实质是数据表示和数据处理。在上述逻辑结构和运算组成的数学模型中，数据还只是机外表示。要完成数据表示和数据处理工作，还要将此模型用计算机程序实现，即还必须研究数据的存储实现和运算实现。存储实现就是把所有数据及其相互关系存储起来，即完成数据的表示工作；运算实现就是在存储结构上完成具体的处理过程并得到完整的程序，即完成数据的处理工作。以上有关概念间的关系可用图 1.3 表示。

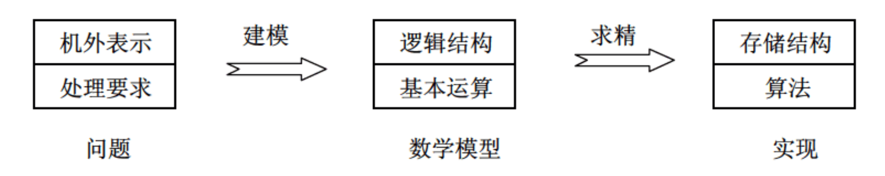


图 1.3　数据结构的主要内容

图 1.3 实际上也是从数据结构的观点来看一个程序的设计过程：首先从具体问题中抽象出一个适当的数学模型，然后设计一个解此模型的算法，最后编程、调试、运行直到最

终获解。其中，寻求数学模型的实质是分析问题，从中提取操作的对象，并找出它们之间的关系，然后加以描述。这个模型对非数值问题，一般不能用数学方程来描述，而是采用诸如线性表、树和图之类的逻辑结构及其运算要求来描述。

图 1.3 也说明了一个程序设计过程是渐进的，体现在两个方面：

（1）数据表示是逐步完成的：机外表示→逻辑结构→存储结构。

（2）数据处理也是逐步完成的：处理要求→基本运算→算法。

其中，数据表示和数据处理是密切相关的，数据处理方式总是与数据的某种相应的表示形式相联系，反之亦然。

对一个数据结构，将其三个方面的内容搞清楚了，也就搞清楚了这个数据结构。另外，这三方面中即使只有某一个不同，我们也将其称做不同的数据结构。例如，线性表是一种逻辑结构，若采用顺序存储方式则称为顺序表；若采用链接存储方式则称为链表。又如，对线性表上的插入、删除运算如果限制在表的一端进行，则称为栈；若插入限制在表的一端进行，删除限制在表的另一端进行，则称队列；更进一步，若栈和队列采用顺序存储方式或链接存储方式，则分别称为顺序栈或链栈、顺序队列或链队列。

数据结构也可用集合论的观点来定义，即将数据结构看成由若干集合组成，如：数据（逻辑）结构是一个二元组(D, R)，其中 D 是具有相同特性的数据元素的有限集，R 是 D 上的关系的有限集。这里的关系可简单地理解为 D 中元素的有效序偶$<d_i, d_j>$集。其中，称 d_j 为 d_i 的（直接）后继，而 d_i 为 d_j 的（直接）前趋。这种表示法有集合论的理论基础，描述严密，易于使用数学方法研究，但不直观，有时也很烦琐，本书没有采用（但对图结构采用了这种描述）。

1.3 算法分析

1.3.1 算法的评价

一般而言，同一个问题可设计出许多不同的算法，这些算法孰优孰劣、如何选择，就涉及算法的评价问题。通常可从下列几个方面评价算法（包括程序）的质量。

（1）**正确性**。指算法应正确实现预定的功能（即处理要求）。关于算法的正确性，一般不进行形式化的证明，而是用测试来验证。这主要是因为实际问题的复杂性，证明起来非常困难。在测试时，用精心选定的若干合法输入来运行算法，检查其结果是否正确。不过，正如著名的计算机科学家 E.Dijkstra 所说的那样，“测试只能指出有错误，而不能指出不存在错误”。

（2）**易读性**。指算法能被理解的难易程度，如思路清晰、层次分明、简单明了等。现在人们比较强调易读性，因为只有算法便于阅读和理解，才便于交流、推广和使用，也便于调试、修改和扩展，以及及早发现隐藏的错误。

（3）**健壮性**。指算法对意外情况（如输入数据非法、内存不足、文件打开失败、读写失败等）能适当地作出反应或进行处理，不会产生不需要的甚至严重的运行结果。比如在输入一个整数时，意外地输入了一个字符，程序会怎样？会不会产生奇怪的结果？

（4）**高效率**。指算法应有较好的时空性能。

这些指标一般很难做到十全十美，因为它们常常互相冲突，故在实际评价中应根据需要有所侧重。在数据结构中主要讨论算法的时空性能，这并不意味着这一指标比其他指标更重要（实际可能恰恰相反），而是课程的性质和内容所决定的。确定一个算法时空性能的工作称为**算法分析**。

算法的时空性能是指算法的时间性能和空间性能，前者指算法的时间耗费，即包含的计算量；后者指算法需要的存储量。算法的时间耗费也称时间复杂性或**时间复杂度**（Time Complexity）；类似，算法的空间耗费也称空间复杂性或**空间复杂度**（Space Complexity）。

同一问题因所采用的数据结构不同，相应算法的效率会有所不同；反之，由算法的执行效率也可反映数据结构的好坏。算法分析是数据结构课程的重要内容之一。

1.3.2 时间复杂度

算法所耗费的时间，应该是算法中所有语句执行时间之和，而每条语句的执行时间是该语句的执行次数（即**频度**，Frequency Count）与该语句执行一次所需时间的乘积。即

$$T=\sum_{\text{语句i}}（\text{频度}\times\text{每次时间}）$$

但算法转换为程序后，每条语句执行一次的时间与所处的软硬件环境有关：

（1）**硬件因素**。主要是机器的指令性能和速度，比如32位机一般比16位机运行快；主频2GHz的机器一般比500MHz的机器快；磁盘的速度一般比磁带快等。

（2）**软件因素**。这又包括语言因素、编译质量、操作系统等。如汇编语言的执行效率一般高于高级语言；编译器不同（如TC、VC等），以及不同的编译选项（如优化代码大小和优化代码速度等）得到的代码也会不同。操作系统对执行时间也有影响，如一般Windows系统下的执行效率快于DOS系统等。

这些因素在一般情况下是很难确定的，所以我们难以算出算法的**绝对时间**，并且这个绝对时间随着运行环境的不同也不同。这使得用绝对时间来评价算法的效率既不合适，也没有意义①。为了突出算法本身的性态而排除算法外的其他因素（即软硬件因素），这里假定每条语句只是抽象地运行，不依赖于某个具体的计算机软硬件环境，并由此进一步假定每条语句执行一次的时间均是单位时间。于是，一个算法的时间耗费就是该算法中所有语句的频度之和。这样，就可以独立于具体机器的软硬件条件来分析算法本身的时间耗费了。为了区别，不妨把这种时间称为**逻辑时间**。

需要注意的是，这里的"语句"，指的是描述算法的基本语句，它的执行，在语法意义上讲应是不可再分割的。换句话说，循环语句的整体、函数调用语句等就不能算作基本语句，因为它们还包括由多个语句组成的循环体或函数体。

例1.3 计算x^{32}。

① 这是指对绝对时间进行**事前估计**。另一种方法是**事后统计**，即测试算法在实际计算机上的运行时间。这可考察算法在具体计算条件下的实际性能及不同算法间实际性能的差异，但在计算条件变化后这些结果会有变化。下述逻辑时间的分析属于事前估计，但也可在实际运行时对有关语句进行统计，即事后统计（参见附录C排序算法的时间统计）。

解：如果直接计算 x^{32}=x*x*…*x，则要做 31 次乘法。

如果考虑到 $x^{32}=((((x^2)^2)^2)^2)^2$，即每次在平方（自乘）的基础上再平方，则只需要 5 次乘法。

又如，计算 x^{31}，它可通过 x^{32}/x 得到。若不允许用除法，则不能直接用连续平方的方法，如果考虑 $x^{31}=x(x(x(x(x)^2)^2)^2)^2$，则也只需 8 次乘法，而不是 30 次。

可见，对同一问题，不同算法的效率可能是不同的，甚至相差很大。

例 1.4 计算两个方阵的乘积 $C_{n\times n}=A_{n\times n}\times B_{n\times n}$。

解：

```
for(i=0;i<n;i++)                              //n+1
   for(j=0;j<n;j++) {                         //n(n+1)
      c[i][j]=0;                              //n^2
      for(k=0;k<n;k++)                        //n^2(n+1)
         c[i][j]=c[i][j]+a[i][k]*b[k][j];     //n^3
   }
```

其中右边列出了各语句的频度。注意，对语句 for(i=0; i<n; i++)等[①]，虽然它控制循环体执行了 n 次，但它自己却执行了 n+1 次。这是因为循环条件 i<n 检测了 n+1 次：前 n 次检测时该条件成立，最后一次检测时该条件不成立（此时 i=n，循环结束）。算法的时间耗费为各语句的频度之和：

$$T(n)=(n+1)+n(n+1)+n^2+n^2(n+1)+n^3=2n^3+3n^2+2n+1 \tag{1.1}$$

可见，它是矩阵阶数 n 的函数。

一般地，我们将问题输入数据量（或初始数据量）的大小，或与之相关的某个量，称为问题的**规模**（Size，大小），并用一个整数表示。例如，矩阵乘积问题的规模是矩阵的阶数，而一个图论问题的规模则是图中的顶点数或边数。一个算法的时间复杂度可以描述为输入规模的函数（空间复杂度也是如此）。在以后的讨论中，我们一般用 T(n)表示时间复杂度、S(n)表示空间复杂度，其中 n 为问题的规模。

时空复杂度的准确表示常常是件非常困难的事，因为很多算法的时间复杂度难以给出解析形式，或者非常复杂。然而，我们看到，当问题规模较大时，复杂度表示式中实际上只有一些占主导地位的项有意义，当 n 较大如 n=100 时，式(1.1)中只有高次项 $2n^3$ 有意义，其他低次项可忽略不计。所以，人们往往放弃寻求确切的时空复杂度函数的企图，而通常用某些简单函数来近似表示其大致性能，这就是时空复杂度的渐近表示。

当问题的规模 n 趋向无穷大时，我们把时间复杂度 T(n)的数量级（阶）称为算法的**渐近时间复杂度**（Asymptotic Time Complexity）。例如，上述矩阵乘法的时间复杂度 T(n)当 n 趋向无穷大时，显然有

$$\lim_{n\to\infty} T(n)/n^3 = \lim_{n\to\infty}(2n^3+3n^2+2n+1)/n^3 = 2 \tag{1.2}$$

这表明，当 n 充分大时，T(n)和 n^3 之比是一个不等于零的常数，即 T(n)和 n^3 是同阶的，

① 从 C/C++语法上看，单独的 for 并不是“语句”，它与后面的循环体一起才构成“循环语句”。若想象把其执行过程拆开：赋初值 1 次（i=0），条件判断 n+1 次（i<n），循环变量变化 n 次（i++），则该部分频度为 2n+2，但最终复杂性的量级不变。显然这种太细节化的处理既烦琐也不利于突出算法本身的主体。这里取 n+1 相当于只考虑其中频度最大的部分，或者从其实际作用看，把 for 当做一个“循环控制语句”，其控制（即条件判断）了 n+1 次。

或者说 T(n)和 n^3 的数量级相同，记作 $T(n)=O(n^3)$，其中大写字母 O 表示 Order，即数量级。可见，上述矩阵乘法的渐近时间复杂度为 $O(n^3)$。关于大“O”记号，其数学定义是：

设 T(n)和 f(n)是两个非负函数，如果存在正常数 c 和 n_0，使得当 $n \geqslant n_0$ 时都有 $T(n) \leqslant c \cdot f(n)$ 成立，则 T(n)=O(f(n))。

根据极限定义知，f(n)是 T(n)的一个上界函数（当 n 足够大时）。一般而言，T(n)的上界函数可能很多，但我们一般取一个形式简单且较接近的上界。比如 2n+3=O(n)，显然也有 $2n+3=O(n^2)$，甚至 $2n+3=O(2^n)$，但只有最接近的 O(n)才更有意义。另外，渐近分析中一般忽略数量级中的系数，既可简化分析又可突出重点。

类似，也可以定义算法的下界函数 g(n)，并用大“Ω”表示为 $T(n)=\Omega(g(n))$。同样，下界函数也可能很多，一般也取一个形式简单且较接近的下界。当上、下界相等时，还可用大“Θ”表示，如 T(n)=O(h(n))且 $T(n)=\Omega(h(n))$时，则 $T(n)=\Theta(h(n))$。但本书并没有这样严格区分，一则为了简便，二则很多算法的上、下界相等，三种表示法是一致的。

采用渐近复杂度表示可突出算法的本质问题，并可忽略一些语句书写细节上的差异。如“x=y; y=z;”和“x=a+b; y=c+x;”分别是 2 条语句，但写成“x=y, y=z;”和“y=c+(x=a+b);”则分别是 1 条语句。显然这些细节差异或变形并不影响复杂度的量级。

由式 1.2 可见，渐近时间复杂度实际上由频度最大的语句（高次项）决定，故在具体分析时，可作如下处理：

（1）若语句很少执行，且与规模无关，则可忽略不计。

（2）若所有语句都与规模无关，即使有上千条语句，其执行时间也不过是一个较大的常数，时间复杂度也只是 $O(n^0)=O(1)$。

（3）一般可只考虑与程序规模有关的频度最大的语句，如循环语句的循环体，多重循环的内循环等。

例 1.5　交换 a 和 b 的内容。

解：

```
tmp=a;
a=b;
b=tmp;
```

以上三条语句的频度都为 1，与规模 n 无关，立即可知 T(n)=O(1)。

例 1.6　求 n 以内所有 2 的幂次数之和，即 $1+2^1+2^2+\cdots+2^k$，$2^k \leqslant n$。

解：

```
sum=0;
for(i=1;i<=n;i*=2)
   sum+=i;
```

这里循环体的执行次数未知，但显然不是 n 次，设为 k 次，由于 $2^k \leqslant n$，所以 $k=O(\log_2 n)$，从而 $T(n)=O(\log_2 n)$。

例 1.7　将二维数组 A[n][n]的内容清空。

解：

```
for(i=0;i<n;i++)
   for(j=0;j<n;j++)
      A[i][j]=0;
```

显然，频度最大的语句是内循环体 A[i][j]=0，而内循环和外循环各执行 n 次，即该语句共执行了 n×n 次，所以 $T(n)=O(n^2)$。

例 1.8 将二维数组 A[n][n]的下三角部分清空。

解：

```
for(i=0;i<n;i++)
   for(j=0;j<=i;j++)
      A[i][j]=0;
```

这里频度最大的语句仍然是内循环体 A[i][j]=0，其中外循环执行了 n 次，内循环执行了 i+1 次，但该语句总的频度不能简单地算成 n×(i+1)，因为 i 不是常量，应该按循环从内向外累加出来：$\sum_{i=0}^{n-1}(i+1)=1+2+3+\cdots+n=n(n+1)/2$，所以 $T(n)=O(n^2)$。

有些算法是用递归方法描述的，相应地，时间复杂度一般可用递归方程表示，对该方程求解就可得到复杂度的具体表达式，本书在树、排序等章节中就有这方面的例子。

在实际算法分析时，还常常根据问题的特点，选择一种或几种关键操作，如数值计算问题中的乘法和除法、查找问题中的比较、排序问题中的比较和移动等，作为“**标准操作**”，来考察标准操作的时间复杂度。由于各种操作都有对应的语句，这相当于考察特定语句的频度或时间复杂度，分析方法是相同的。

将常见的渐近复杂度，按数量级递增排列，依次为：常数阶 O(1)、对数阶 $O(\log_2 n)$、线性阶 O(n)、线性对数阶 $O(n\log_2 n)$、平方阶 $O(n^2)$、立方阶 $O(n^3)$、…、k 次方阶 $O(n^k)$、指数阶 $O(2^n)$，另外还有复杂度更高的阶乘阶 O(n!)和 n 次方阶 $O(n^n)$等。其中，后三种常统称为**指数复杂性**，其他则统称为**多项式复杂性**。图 1.4 展示了几种复杂度的增长率。

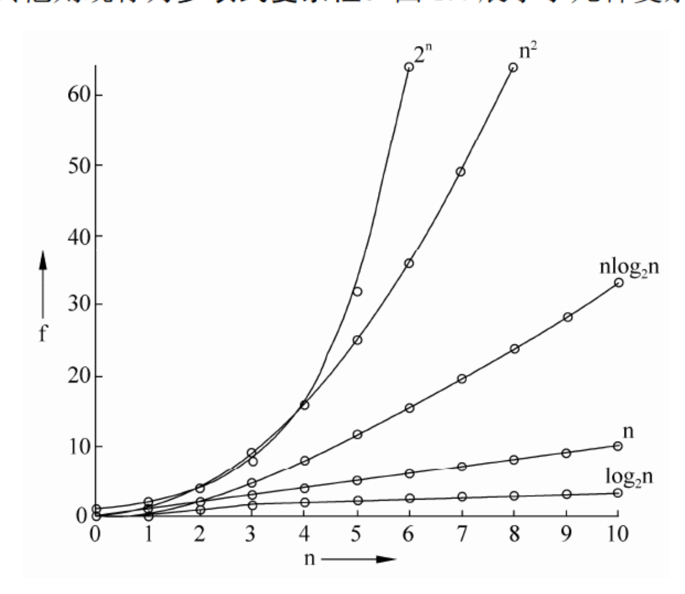


图 1.4 几种常见复杂度的增长趋势

很多算法的时间复杂度不仅与问题的规模有关，还与所处理的数据集的状态有关。这是因为有些语句的频度对算法本身而言是不确定的，要取决于具体的数据情况。通常，这类算法在程序上的一个特点是用 if…else 或 switch 语句来处理不同的情况。对这类问题，原则上需用概率论的方法来解决，但一般是根据数据集中可能出现的最坏或最好情况，估

计出算法的**最坏**（Worst）**时间复杂度**或**最好**（Best）**时间复杂度**；或者对数据集的分布作出某种假定（如等概率），讨论算法的**平均**（Average）**时间复杂度**。

例 1.9　在数组 A[n]中查找值为 K 的元素。

解：

```
for(i=0;i<n;i++)
   if(A[i]==K) break;
```

这里，比较语句的执行次数不仅与问题的规模 n 有关，还与数组 A 中各元素的取值状态有关。最好时，所要查找的元素是数组的第一个元素，只比较 1 次；最坏时，所要查找的元素是数组的最后一个元素，要比较 n 次。如果所要查找的元素出现在数组中各位置的概率是相等的，则不难知道，查找成功时平均比较(n+1)/2 次。

与算法分析类似，还可以进行“问题”分析。即对某问题所有可能的算法，包括尚未发现的算法，分析它们效率的总体上下限。在本书排序一章会提到这个问题。

1.3.3　空间复杂度

一个算法所耗费的空间，应该是算法运行所需的各种存储空间的总和，其中包括代码和数据两部分，但数据结构所讨论的空间复杂度是指数据部分的空间耗费（包括函数调用所需的栈空间）。根据是否与问题的输入有关，这部分空间又可分为固定部分和可变部分。前者与输入数据的状态和规模无关，如程序常量和辅助工作变量等；后者则直接相关。当问题的规模较大时，可变部分会远大于固定部分，所以数据结构中一般讨论问题的渐近空间复杂度，也简称空间复杂度，分析方法与时间复杂度类似，这里就不多述了。

1.3.4　时空复杂度的意义

（1）时间复杂度可用于比较不同算法时间性能的相对好坏。例如，设有算法 1 和算法 2 求解同一个问题，它们的时间复杂度分别是 $T_1(n)=100n^2=O(n^2)$，$T_2(n)=5n^3=O(n^3)$。它们的时间开销之比 $100n^2/5n^3=20/n$，当规模较小时，如 $n<20$，有 $T_1(n)>T_2(n)$，后者花费的时间较少。但是，随着问题规模的增大，算法 2 的时间耗费就会超过算法 1，并且其差距还会继续加大，此时算法 1 要有效得多。

实际上，在评价一个算法的时间性能时，一般采用渐近时间复杂度。另外，往往对算法的时间复杂度和渐近时间复杂度不予区分，经常将渐近时间复杂度简称为时间复杂度。

从图 1.4 可以看到，$O(n)$与 $O(n\log_2 n)$ 的增长比较平缓，所以实际应用中 $O(n)$与 $O(n\log_2 n)$ 的算法差别可能并不很大，但 $O(n\log_2 n)$ 和 $O(n^2)$的算法差别就非常大了。

（2）时间复杂度可从宏观上评价算法的时间性能。例如，指数阶 $O(2^n)$的复杂度增长太快，当 n 稍大时效率极低，无法应用，即这类算法是不可行的。假设计算机每秒执行 1000G 条指令（目前微机为 3G 左右），则当 n=100 时，执行 2^n 条指令需耗时：

$$T=\frac{2^{100}\ /(1000\times10^9)}{365\times24\times3600}\approx4\times10^{10}\text{年}$$（即约 400 亿年！）

不难理解，如果能将现有指数时间算法中的任何一个化简为多项式时间算法，将是一个伟大的成就。

（3）时间复杂度可粗略估计同一个算法的时间变化趋势。如高斯消元法解线性方程组的时间复杂度为 $O(n^3)$，即求解时间随问题规模呈立方增长，如果阶数增加一倍，求解时间大约将是原来的 $2^3=8$ 倍。于是，假设在某机器上解 100 阶的线性方程组用了 5 分钟，则解 200 阶的线性方程组将大概用 $5\times8=40$ 分钟！

（4）平均时间复杂度反映的是总体性能，比较符合实际使用情况；最好时间复杂度反映的是理想情况，最坏时间复杂度反映的是最坏会怎么样，它们发生的概率一般都较小。在后两者中，一般较关心最坏时间复杂度，因为它太差的话，即使发生的概率小之又小，则给人的感觉就是不可靠，这时即便最好或平均情况再好，人们也难以采用。

（5）对实际问题，一般时间复杂度显得比空间复杂度重要些。这主要是因为实际问题的规模一般较大，计算时间较长，常常是应用中的一个突出问题。也许采用更快的计算机（比如不久后计算机技术发展了，或直接采用当前更高速的计算机等）可以解决原来时间复杂度大的问题，但拥有更好的计算条件后，人们往往又希望求解更大规模或更复杂的问题，其结果是计算机速度的提高总是难以满足实际问题，特别是大规模复杂问题的要求。

比如，大多数问题的时间复杂度都高于 $O(n)$，常见的是 $O(n\log_2 n)$、$O(n^2)$、$O(n^3)$ 等，以一般线性方程组求解的时间复杂度 $O(n^3)$ 为例，假设计算机速度提高到了原来的 10 倍，则相同时间内可求解的问题规模大约只能提高到原来的 $\sqrt[3]{10}\approx2.15$ 倍。并且不难看到，时间复杂度越高的算法从提高机器速度得到的收益相对就越小。所以追求高速算法总是一件重要事情。相比起来，空间复杂度问题就没有时间复杂度那样严重，但这并不是因为计算机的存储空间是海量的，而是由实际问题的本质决定的。

（6）时间复杂度与空间复杂度往往是一对矛盾，常常可以用空间换取速度，反之亦然。也就是说，为了获得较快的速度，一般要花费较多的空间；为了使用较少的空间，一般要花费较多的时间。这方面的例子在数据结构的一些算法中屡见不鲜。但应指出：

① 时间和空间的这种彼消此长的作用，一般只是在原有基础上某个倍数或比率的变化，并不能改变算法本身复杂度的量级，比如，原复杂度为 $O(\alpha n)$，新复杂度为 $O(\beta n)$，只是其中的系数不同而已。

② 这个结果是对内存而言的，如果是外存，情况可能正好相反，因为外存速度一般比内存和 CPU 速度慢几个数量级，外存用得越大，数据输入和输出所花费的时间就越大，整个处理时间一般会越大，而不是越小。

（7）有助于正确认识和运用代码调整与优化。有时对代码进行调整（Code Tuning）可使程序的执行时间大为缩短（如减少到原来的 1/10），或者降低存储需求（如降到原来的 1/2），但这也不能改变算法复杂度的量级，因为算法的复杂度是由算法本身决定的。代码调整也不能代替算法分析的作用。事实上，只有在算法分析的基础上才知道哪些地方是调整的关键，即那些频度最大或空间需求最大的语句，反之，对那些只占总执行次数很少比率的语句进行调整是没有意义的。另外，调整也不应牺牲算法或程序的可读性。一个较好的方法是利用编译器进行代码优化，不过它只对表达式、循环、跳转等优化比较有效，而对算法本身作用不大，因为编译器不可能改变算法。

所以，要获得最好的时间和空间复杂度，根本还在于采用更好的数据结构和算法。采用高速计算机、代码调整或优化等，只能作为辅助手段。

习 题 一

1.1 若没有计算机，是否就没有数据结构的问题？算法是否最终都要转换为计算机程序？

1.2 解释数据、数据元素、数据项、逻辑结构、存储结构、运算、算法、程序等概念及其相互关系。

1.3 数据的存储结构是否只有 4 种？

1.4 计算机的速度越来越快，储存量也越来越大，那么研究算法的复杂度有何意义？

1.5 按数量级从小到大的顺序排列下列函数：

2^{100}, $(3/2)^n$, $(2/3)^n$, $n^{3/2}$, $n^{2/3}$, n^n, $n!$, 2^n, $\log_2 n$, $n\log_2 n$

1.6 判断下列程序段的复杂度：

（1）

```
i=1;
while(i*i<=n) i++;
```

（2）

```
s=0;
for(i=1;i<=n;i++)
   for(j=i*i;j<=n;j++)
      s=s+1;
```

1.7 编写算法实现以下功能，并分析其时间复杂度：

（1）求 n 以内所有整数之和，即 1+2+…+n。

（2）求 n 以内所有奇数之和，即 1+3+…+(2k−1)，2k−1≤n。

（3）将 n 以内整数依次相加，但最多加到和为 n，即 1+2+3+…≤n。

1.8 编写算法，将二维数组 A[1..n][1..n]中第 1, 2, 4, 8, …行对角元之前的元素清空。分析其时间复杂度。

1.9 对下列程序段进行代码调整以改善其性能：

```
min=A[0];
for(i=1;i<n;i++)
   if(A[i]<min) min=A[i];
max=A[0];
for(i=1;i<n;i++)
   if(A[i]>max) max=A[i];
```

1.10 能否不用中间量（附加空间）交换两个变量的内容？

第2章 线性表

线性表是最简单、最常见的一种数据结构。本章将详细介绍线性表的基本概念、线性表的两种主要存储结构——顺序表和链表、线性表的一些常见运算及其在这两种存储结构上的实现。线性表的有关知识也有助于下一章将要讨论的栈、队列、串等数据结构。

2.1 线性表的基本概念

线性表是将一批数据元素一个接一个地依次排列得到的一种结构，这方面的例子不胜枚举。例如，英文字母表(A, B, C, …, Z)就是一个线性表，表中的每一个英文字母是一个数据元素；又如，一副扑克牌的点数表(2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A)也是一个线性表，其中每一张牌的点数是一个数据元素。在较为复杂的线性表中，数据元素可由若干数据项组成，如学生成绩表，每个学生的所有信息组成一个数据元素，它由学号、姓名、班级、各科成绩等数据项组成。综合类似例子，可以将线性表一般地描述为：

线性表（Linear List）是由 n（n≥0）个数据元素（结点）$a_1, a_2, \cdots, a_n$ 组成的有限序列。其中，数据元素的个数 n 定义为表的长度。当 n=0 时称为空表，记作( )或 ∅，若线性表的名字为 L，则非空的线性表（n>0）记作：

$$L=(a_1, a_2, \cdots, a_n)$$

这里数据元素 a_i（$1\leqslant i\leqslant n$）只是一个抽象的符号，其具体含义在不同情况下可以不同，但同一个线性表的数据元素类型一般要求相同，这称为**同构**（Homogeneity）。

线性表的相邻元素之间存在着前后顺序关系，其中第一个元素无前趋，最后一个元素无后继，其他每个元素有且仅有一个直接前趋和一个直接后继。可见，线性表是一种线性结构。

设 L 代表某线性表，对线性表的基本运算，常见的有以下几种。

（1）初始化 INITIATE(L)：加工型运算，作用是建立一个空表 L=∅。执行该操作后，线性表的其他操作才能进行。

（2）求表长 LENGTH(L)：引用型运算，结果是线性表 L 中的结点个数。

（3）读表元（按序号查找）GET(L, i)：引用型运算，若 1≤i≤LENGTH(L)时，结果是表 L 中的第 i 个（序号为 i）结点（值或地址）；否则，结果为一个特殊值。

（4）定位（按值查找）LOCATE(L, x)：引用型运算，当线性表 L 中存在一个或多个值为 x 的结点时，结果是这些结点中首次找到的结点（序号或地址）；否则，结果为一个特殊

值（如零）。

（5）插入 INSERT(L, x, i)：加工型运算，在线性表 L 的第 i 个位置插入一个值为 x 的新结点，使得原表由(a_1, …, a_{i-1}, a_i, …, a_n)变为(a_1, …, a_{i-1}, x, a_i, …, a_n)。这里 $1 \leqslant i \leqslant n+1$，而 n 是原表的长度。插入后元素个数增 1，原第 i 个元素之后的元素的序号也分别增 1。

（6）删除 DELETE(L, i)：加工型运算，删除线性表 L 的第 i 个结点，使得原表由(a_1, …, a_{i-1}, a_i, a_{i+1}, …, a_n)变为(a_1, …, a_{i-1}, a_{i+1}, …, a_n)。这里 $1 \leqslant i \leqslant n$，而 n 是原表的长度。删除后元素个数减 1，原第 i 个元素之后的元素的序号也分别减 1。

线性表的其他运算可用这 6 种基本运算来实现，如修改第 i 个元素的值 SET(L, i, x)、求某个元素的前趋 PRIOR(L, x)和后继 NEXT(L, x)、判断线性表是否为空 EMPTY(L)、线性表的合并 MERGE(L, L1, L2)和拆分 SPLIT(L, i, L1, L2)等。另外，排序也可看成基本运算，但由于其特殊性我们在第 7 章单独讨论。注意，实际问题并不一定需要同时执行以上运算，要根据情况进行选择。

由于数据的运算是定义在逻辑结构上的，而运算的具体实现则是在存储结构上进行的，所以，线性表的上述运算，只是在逻辑结构上给出了运算的功能是“做什么”，至于“如何做”等实现细节，只有确定了存储结构之后才能考虑。

2.2 线性表的顺序实现

在本书中，一种数据结构的顺序实现是指按顺序存储方式构建其存储结构，并在此存储结构上实现其基本运算。

2.2.1 顺序表

将一个线性表存储到计算机中，可以采用许多不同的方法，其中既简单又自然的是顺序存储方法，即把线性表的结点按逻辑次序依次存放到一组地址连续的存储单元里，用这种方法存储的线性表简称为**顺序表**（Sequential List[①]）。

假设顺序表中每个结点占用 c 个存储单元，其中第一个结点 a_1 的存储地址（以下简称基地址）是 Loc(1)，则结点 a_i 的存储地址 Loc(i)可通过下式计算：

$$Loc(i)=Loc(1)+(i-1)\times c \qquad 1 \leqslant i \leqslant n$$

也就是说，在顺序表中，每个结点 a_i 的存储地址是该结点在表中的位置 i 的线性函数，只要知道基地址和每个结点的大小，就可在相同的（逻辑）时间内求出任一结点的存储地址。因此顺序表是一种随机存取结构。

在程序设计语言中，一维数组（本书也称**向量**）一般都是用顺序存储表示的，故可用一维数组来描述顺序表。但数组定义后其大小一般不能再改变，而线性表的表长是可变的（如插入和删除时），所以要将数组预设足够的大小（容量）；同时还需要一个变量指出线性表在数组中的当前状况，如元素的个数或最后一个元素在数组中的位置等。这两方面的信

① 有的文献用 Sequential List 表示线性表，用 Contiguous List 表示顺序表。

息共同描述一个顺序表，可将它们封装在一起。

对 C/C++语言，顺序表可定义如下：

```
typedef int datatype;        //线性表结点的数据类型，假设为 int
const int maxsize=100;       //线性表可能的最大长度，这里假设为 100
typedef struct {
  datatype data[maxsize];    //线性表的存储向量，第一个结点是 data[0]
  int n;                     //线性表的当前长度
} sqlist;                    //顺序表类型
```

其中，

（1）数据域 data 是存放线性表各结点的数组空间，下标范围是 0～maxsize–1，线性表的结点 a_i 存放在数组元素 data[i−1]中。显然线性表结点的个数不能超过数组空间的大小 maxsize。

（2）数据域 n 记录线性表当前的长度，终端结点的数组下标为 n–1。

（3）datatype 是线性表结点的类型，它应是某种定义过的类型，具体含义要视实际情况而定。例如，若线性表是英文字母表，则 datatype 就是字符类型 char；若线性表是学生成绩表，则 datatype 就是已定义过的表示学生情况的结构类型。

（4）顺序表类型 sqlist 是一个结构类型，它将顺序表的有关信息封装在一起作为一个整体看待，符合结构程序设计的思想。这样每个顺序表只用一个名字就可以表示，但当要访问顺序表的细节时，需要使用成员选择运算符或指针指向运算符。例如设 L 是 sqlist 类型的变量，则顺序表 L 的第一个结点是 L.data[0]，终端结点是 L.data[n−1]。这便于管理程序中有多个顺序表的情况。如果程序中只有个别顺序表，为了书写简便，也可不用结构而分别由一个向量和一个整型变量来表示它。图 2.1 为顺序表的示意图，其中 b 为第一个元素的存储地址。

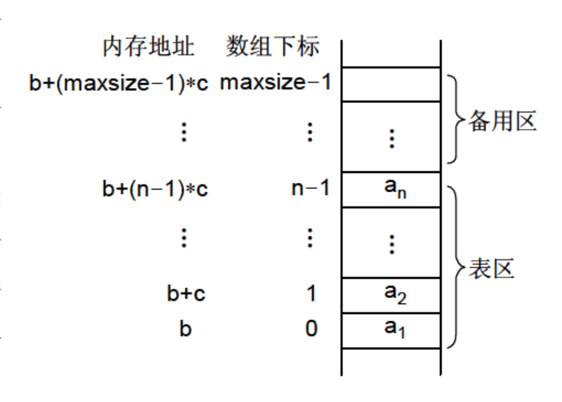


图 2.1 顺序表示意图

总之，顺序表是用向量实现的线性表，向量下标可看成结点的相对地址。它的特点是逻辑上相邻的结点其物理位置亦相邻。

顺便指出，顺序表实现时也可从数组下标 1 开始使用，这时结点 a_i 存放在数组元素 data[i]中，数组大小为 data[maxsize+1]。0 号单元不用，但也可用来存放线性表长度（这时可省略顺序表的长度域 n）。

2.2.2 顺序表上的基本运算

定义了线性表的存储结构之后，就可以讨论在该存储结构上如何实现原来定义在逻辑结构上的运算了。在顺序表中，线性表的有些运算很容易实现。例如，初始化 INITIATE(L)就是将 L.n 置为 0；取第 i 个结点 GET(L, i)即取出 L.data[i−1]；求表长 LENGTH(L)即取出 L.n 等，它们的时间复杂度为 O(1)。以下仅讨论插入、删除和定位等运算。

1．插入

线性表的插入运算是指在表的第 i（$1 \leqslant i \leqslant n+1$）个位置上，插入一个新结点 x，使长度为 n 的线性表(a_1, …, a_{i-1}, a_i, …, a_n)变为长度为 n+1 的线性表(a_1, …, a_{i-1}, x, a_i, …, a_n)。

用顺序表作为线性表的存储结构时，由于结点的物理顺序必须和结点的逻辑顺序保持一致，因此在插入时，要先将表中位置 i～n 上的结点全部后移一位以空出第 i 个位置，然后在该位置上插入新结点 x，最后表长加 1。特别地，插入位置 i=n+1 时无须移动结点，直接将 x 插入到表的末尾即可。插入过程见图 2.2。

注意，结点移动的次序只能从后向前进行，即按 n, n−1, …, i 的次序进行移动，否则，若从前向后移动，则后面元素的数据将被前面的冲掉，结果插入点后的数据将全部都是 a_i。

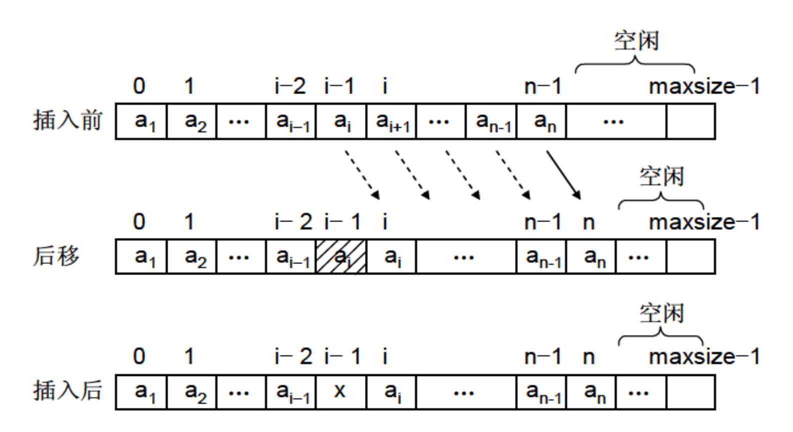


图 2.2　顺序表中插入结点过程示意图

顺序表插入的具体算法如下（其中通过函数的返回值区分插入的执行情况）：

```
int insert(sqlist *L,datatype x,int i) {//将 x 插入到顺序表 L 的第 i 个位置上
  int j;
  if(L->n==maxsize) {cout<<"表满，不能插入!(上溢)\n";return -1;}
  if(i<1 || i>L->n+1) {cout<<"非法插入位置!\n";return 0;}
  for(j=L->n;j>=i;j--)
    L->data[j]=L->data[j-1];         //结点后移
  L->data[i-1]=x;                    //插入 x，第 i 个结点的数组下标是 i-1
  L->n++;                            //修改表长
  return 1;                          //插入成功
}
```

注意函数参数表中，对应顺序表的参数是其指针（传地址）而不是顺序表本身（传值），是考虑到顺序表的内容较大，按值传递比较费时（实参内容要复制到形参）。对此也可采用 C++的引用参数传递（略，参见附录 B）。

设表的长度为 n，上述算法的时间主要花在 for 循环中的结点后移上，该语句的执行次数也就是结点的移动次数，为 n−i+1。可见，结点的移动次数不仅与表的长度 n 有关，还与插入位置 i 有关。当 i=n+1 时，结点移动次数为 0；当 i=1 时，结点移动次数为 n。即算法的最好时间复杂度是 O(1)，最坏时间复杂度是 O(n)。

下面考察算法的平均时间性能。设在表中第 i 个位置上插入一个结点的概率为 p_i，在第 i 个位置上插入一个结点时的移动次数为 c_i，则结点的平均移动次数为 $M=\sum_{i=1}^{n+1} p_i c_i$，其

中 c_i=n−i+1，但 p_i 未知。不失一般性，假设在表中任何合法位置（1≤i≤n+1）上插入结点的机会是均等的，则 $p_1 = p_2 = \cdots = p_{n+1} = 1/(n+1)$，因此，在等概率插入的情况下：

$$M = \sum_{i=1}^{n+1} p_i c_i = \sum_{i=1}^{n+1} \frac{n-i+1}{n+1} = \frac{n}{2}$$

也就是说，在顺序表上做插入运算，平均要移动表中一半的结点。当表长 n 较大时，算法的效率相当低。M 中的系数较小，但就数量级而言，它仍然是线性阶的，因此算法的平均时间复杂度是 O(n)。

2. 删除

线性表的删除运算是指将表的第 i（1≤i≤n）个结点删去，使长度为 n 的线性表(a_1, …, a_{i-1}, a_i, a_{i+1}, …, a_n)变为长度为 n−1 的线性表(a_1, …, a_{i-1}, a_{i+1}, …, a_n)。

和插入算法类似，在顺序表上实现删除运算也必须移动结点：先将表中位置 i+1～n 上的结点全部前移一位以填补删除操作造成的空缺，然后表长减 1。特别地，删除位置 i=n 时无须移动结点，直接删除终端结点即可。

注意，这里结点的移动次序只能从前向后进行，即按 i+1, i+2, …, n 的次序进行。删除过程见图 2.3。

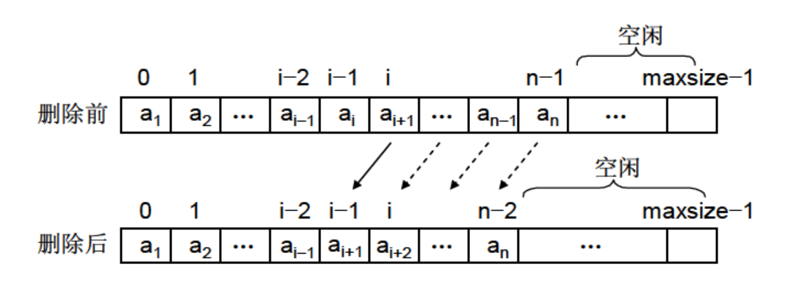


图 2.3 顺序表中删除结点过程示意图

删除过程的具体算法如下：

```
int delete(sqlist *L,int i) {        //从顺序表中删除第 i 个位置上的结点
  int j;
  if(L->n==0) {cout<<"表空，不能删除!(下溢)\n";return -1;}
  if(i<1 || i>L->n) {cout<<"非法删除位置!\n";return 0;}
  for(j=i+1;j<=L->n;j++)
    L->data[j-2]=L->data[j-1];       //结点前移，第 j 个结点的数组下标是 j-1
  L->n--;                            //修改表长
  return 1;                          //删除成功
}
```

该算法的时间分析与插入算法类似，结点的移动次数也是由表长 n 和位置 i 决定的。删除第 i 个结点时，结点的移动次数为 c_i=n−i。当 i=n 时结点移动次数最少，为 0 次；当 i=1 时结点移动次数最多，为 n−1 次。这两种情况下算法的时间复杂度分别是 O(1)和 O(n)。

删除算法的平均性能分析也与插入算法相似。如果在每个位置 i（1≤i≤n）上删除结点的概率相等，则 $p_i = 1/n$ 。这样，在等概率删除的情况下：

$$M = \sum_{i=1}^{n} p_i c_i = \sum_{i=1}^{n} \frac{n-i}{n} = \frac{n-1}{2}$$

即在顺序表上做删除运算，平均要移动表中约一半的结点，平均时间复杂度也是 O(n)。

3. 定位

定位运算 LOCATE(L, x)的功能是求表 L 中第一个值为 x 的结点的序号，当不存在这种结点时结果为 0。因此可从前向后比较各结点的值是否等于 x。算法如下：

```
int locate(sqlist *L,datatype x) {
  int i;
  i=1;
  while(i<=L->n && L->data[i-1]!=x) i++;
  if(i<=L->n) return i;
  else return 0;
}
```

其中 while 循环结束时有两种情况，一是找到了值为 x 的结点，此时 i 为有效结点号，i<=L–>n；一是表中没有值为 x 的结点，此时循环终止于 i=L–>n+1。故最后分情况返回不同结果。但这两种情况也可用取模运算统一起来：return i%(L–>n+1)。

显然，上述算法的时间主要花在结点值的比较上，易知比较次数最少 1 次，最多 n 次，平均(n+1)/2 次，所以其最好、最坏、平均时间复杂度也分别为 O(1)、O(n)和 O(n)。

顺便指出，对于顺序表中插入和删除时结点的大量移动问题，在 C/C++语言中其实有一个较好的处理方法，就是将顺序表看成串，采用快速的串移动函数。具体就不细述了。

例 2.1　已知顺序表中各结点的值有正有负，试设计算法使负值结点位于顺序表的前面部分，正值结点位于顺序表的后面部分。

解：对此问题一个很自然的想法是，对顺序表从前向后搜索（或称**扫描**），遇到正值结点就将它插入到表的后部（或从后向前搜索，遇到负值结点就将它插入到表的前部）。但我们知道，在顺序表上插入或删除不方便，要移动大量结点，效率不高；而对本问题，这种插入要多次进行，易见移动次数最坏为 $O(n^2)$。

这里我们给出一个由表的两端向中间交替扫描的算法，即先从前向后扫描，找到一个值为正的结点，再从后向前扫描，找到一个值为负的结点，然后交换两者的内容。接着对剩余部分继续进行同样的过程，直到两个扫描方向相遇，整个表扫描处理完毕。这样就避免了大量结点的移动问题，算法如下：

```
void order(sqlist *L) {
  datatype x;
  int i,j;
  i=0;j=L->n-1;
  while(i<j) {
    while(i<j && L->data[i]<0) i++; //从前向后扫描
    while(i<j && L->data[j]>0) j--; //从后向前扫描
    if(i<j) {                      //交换
      x=L->data[i];L->data[i]=L->data[j];L->data[j]=x;
      i++;j--;                     //调整扫描范围
    }
  }
}
```

显然该算法的时间复杂性为 O(n)。另外，从前后两端分别进行的扫描可以并行处理，但我们使用的计算机一般是串行工作的，所以只能交替进行。

例 2.2　已知顺序表中有若干值为零的结点，试设计算法删除这些零值结点。要求不改

变其他结点的相对位置。

解：与例 2.1 类似，对此问题一个很自然的想法是：单向扫描，对顺序表从前向后扫描，遇到零值结点就删除之，即将其后的所有结点都前移一位。显然这要多次引起大量结点的移动，易见最坏复杂性为 $O(n^2)$。

若采用例 2.1 的算法：交替扫描，从前向后扫描，找到一个零值结点，再从后向前扫描，找到一个非零值结点，然后两者交换。这时复杂性虽然为 O(n)，但会改变结点间的相对位置。

这里给出一个改进的单向扫描算法：每扫描到一个零元，并不马上删除它（即不调整其他结点的位置），而是累计当前的零元数 s；每扫描到一个非零元，就将其前移 s 个位置。这时每个非零结点最多移动 1 次，复杂性为 O(n)，且不改变它们的相对位置。算法如下：

```
void purge(sqlist *L) {
  int i,s;
  s=0;
  for(i=0;i<L->n;i++)
    if(L->data[i]==0) s++;                    //累计零元数
    else if(s>0) L->data[i-s]=L->data[i];     //非零元前移 s 位
  }
  L->n=L->n-s;                                //调整表长
}
```

以上介绍了双向扫描和单向扫描的两个典型算法，它们是一些较复杂算法的一个基础（如以后将要介绍的快速排序算法）。有些问题这两种扫描方法要有选择地使用（如上面的两个例子），但有些问题这两种方法都可，区别不大。

例 2.3 将顺序表中各结点逆置，即 a_1 和 a_n 互换、a_2 和 a_{n-1} 互换等。

解：互换的一般规律是 a_i 和 a_{n-i+1} 互换，其中 $1\leqslant i<n/2$，这可通过一个从前向后的单向扫描来实现：

```
for(i=1;i<n/2;i++) 交换 a_i 和 a_{n-i+1};
```

这里循环上限 n/2 和目标位置 a_{n-i+1} 的下标 n−i+1 均要计算出来。如果采用双向扫描就可避免这个问题，并且算法也很简洁：

```
for(i=1,j=n;i<j;i++,j--) 交换 a_i 和 a_j;
```

当然也可采用 while 循环（略，对本例没有 for 循环简洁）。

2.3 线性表的链接实现

由上节的讨论可知，线性表的顺序表表示，其特点是用物理位置上的邻接关系来表示结点间的逻辑关系，这一特点使得顺序表有如下的优缺点。

其优点是：

（1）无须为表示结点间的逻辑关系而增加额外的存储空间。

（2）可以方便地随机存取表中任一结点。

（3）方法简单，各种高级语言中都有数组类型，容易实现。

其缺点是：

（1）插入或删除运算不方便，除表尾的位置外，在表的其他位置上进行插入或删除操作都必须移动大量的结点，效率较低。

（2）需要预先分配（静态分配）足够大的连续存储空间。若分配过大，则顺序表后面的空间可能长期闲置而得不到充分利用；若分配过小，又可能在使用中因空间不足而造成溢出。

为了克服顺序表的缺点，可以采用链接方式存储线性表，通常将链接方式存储的线性表称为（线性）**链表**（Linked List）。链表是用一组任意的存储单元来存放线性表的结点，这组存储单元既可以是连续的，也可以是不连续的，甚至是零散分布在内存中的任何位置上。因此，链表中结点的逻辑次序和物理次序不一定相同。为了能正确表示结点间的逻辑关系，在存储每个结点值的同时，还必须存储其后继或前趋结点的地址（或位置）信息，这个信息称为**指针**（pointer）或**链**（link）。链表正是通过结点的链域将线性表的各个结点按其逻辑顺序链接在一起的。简单地说，链表就是用指针表示结点间的逻辑关系。

本节以下主要从两个角度来讨论链表：根据链表总空间构成的特点，将链表分为动态链表和静态链表；根据指针链接方式的不同，将链表分为单链表、循环链表和双链表。其中重点讨论单链表。特别指出的是，链接存储是最常用的存储方法之一，它不仅可用来表示线性表，而且可以用来表示各种非线性的数据结构，在以后的各章节中将反复使用。

在本书中，一种数据结构的链接实现是指按链式存储方式构建其存储结构，并在此存储结构上实现其基本运算。

2.3.1　单链表

在单链表中，每个结点由两部分信息组成，一个是数据域，用来存放结点的值（内容）；另一个是指针域（亦称链域），用来存放结点的直接后继的地址（或位置）。所有结点通过指针域链接在一起，构成一个链表，其中每个结点只有一个指针域，故称之为**单链表**（Singly Linked List）。数据域和指针域一般用 data 和 next 表示，结点结构为：

data	next

单链表每个结点的地址存放在其前趋结点的 next 域中，但开始结点无前趋，故应另外用一个指针来指向它，这个指针称为**头指针**，存放这个指针的变量称为头指针变量，一般用 head 表示。另外，终端结点无后继，它的指针域为空，即 NULL（图示中常用∧表示）。例如，图 2.4 是线性表(75, 03, 26, 78, 90, 55)的单链表示意图，这里假设指针和数据各占两个字节的存储空间。

头指针

head　1100

存储地址	数据域	指针域
⋮	⋮	⋮
1002	26	1104
⋮	⋮	⋮
1100	75	1206
1104	78	1210
⋮	⋮	⋮
1206	03	1002
1210	90	1234
⋮	⋮	⋮
1234	55	NULL
⋮	⋮	⋮

图 2.4　单链表存储状态示意图

由于单链表只注重结点间的逻辑顺序，并不关心每个结点的实际存储位置，因此通常用箭头来表示指针域中的指针，从而将链表简洁直观地画成用箭头链接起来的结点序列。例如图 2.4 所示单链表

可以画成图 2.5 的形式。

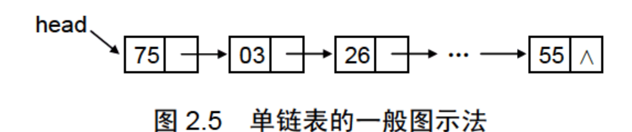


图 2.5 单链表的一般图示法

如果给定了头指针，也就知道了单链表第一个结点的位置，于是沿着指针链就可以“顺藤摸瓜”地找到表中任一结点；而表中任一结点也只有通过指向它的指针才能访问，所以头指针变量具有标识单链表的作用，或者说，单链表由头指针唯一确定。因此单链表可以用头指针变量的名字来命名，如图 2.5 所示的单链表就称为表 head 或 head 表。

单链表的类型定义如下：

```
typedef int datatype;           //结点数据类型，假设为 int
typedef struct node * pointer;  //结点指针类型
struct node {                   //结点结构
  datatype data;
  pointer next;
};
typedef pointer lklist;         //单链表类型，即头指针类型
```

其中，

（1）pointer 是指向 struct node 类型变量的指针类型。

（2）struct node 是结构体类型，规定一个结点是由两个域（data 和 next）组成的记录（每个域实际上相当于一个变量），其中 data 是结点的数据域，next 是结点的指针域，后者类型为 pointer（即 struct node *）。

（3）lklist 是一个与 pointer 相同的类型，但名字不同。以后我们用 lklist 来说明头指针变量的类型，也即用来作为单链表的类型；用 pointer 来说明单链表一般结点（包括工作结点）的指针类型。这样，对于说明语句“lklist A;”，我们想到的是 A 为单链表的头指针，也即 A 表示一个单链表；而对于说明语句“pointer A;”，我们想到的是 A 为一个普通结点的地址。一般地，同一类型取不同名字可用来说明类型相同但作用不同的变量，即把类型赋予一定的“语义”，有利于提高程序的可读性。

指针的概念是链式存储结构的核心。要正确区分指针变量、指针、指针所指的结点、结点变量和结点的内容（结点的值）等几个密切相关的不同概念。假设 p 是一个 pointer 类型的指针变量，则

（1）p 的值是一个指针。但若 p 未初始化或未赋过值，则 p 的值无意义，或称 p 无值。

（2）该指针若为 NULL，则不指向任何结点；否则是某个 struct node 类型结点的地址，或者说，该指针指向该结点。这个结点用*p 来标识，其值是结点的内容。注意，单链表的任何结点都只能用指向它的指针变量来标识。

（3）链表的结点空间一般都是动态分配的，所以通常 p 所指的结点变量并不在程序的变量说明部分显式地定义（故没有名字），而是在程序执行过程中，当需要时才临时产生，故称为动态变量。它通过 C++的 new 运算符产生，即

```
p=new node;
```

上式分配一个 node 大小的连续字节空间，返回一个指向该地址的指针，并将该指针存

入指针变量 p 中。

严格说来，用 new 运算符动态申请内存空间时应检测申请是否成功，即该空间指针是否为 NULL，如为 NULL 则系统已无可分配的空间（如内存耗尽）。但如果程序的数据量不大，一般不会出现这种情况，所以，为了简便起见，本书一般不考虑空间不足的情况。

一旦 p 所指的结点变量不再需要了，就可通过 C++的 delete 运算符释放其所占空间：

```
delete p;
```

因此，我们无法通过预先定义的标识符去访问这种动态的结点变量，而只能通过指针 p 来访问它，即用*p 作为该结点变量的名字来访问。注意，如果 p 值为空，则它不指向任何结点，这时，通过*p 来访问结点意味着访问一个不存在的变量，从而引起程序错误。

（4）p 所指向的结点类型是结构（记录）类型，要访问它的成员可用成员选择运算符：(*p).data、(*p).next 或指针指向运算符：p->data、p->next。

有关指针的详细知识，请参考 C/C++语言的有关资料。

以下为叙述方便，将不再严格区分指针变量和指针这两个概念，如将头指针变量称为头指针，将修改某指针变量的值称为修改某指针等。

2.3.2　单链表上的运算

下面讨论单链表上的一些运算实现，注意，这里并不限于线性表的几种基本运算。

由于链表的结点空间是动态分配的，所以我们不声明结点变量本身，而是声明指向结点变量的指针，然后在需要时，用运算符 new 分配结点空间；相应地，在结点不再有用时要用运算符 delete 释放结点空间，特别地，在程序结束时应该将链表的所有结点空间释放，故需要增加一个销毁运算。

在这里还要特别注意链表运算的一个基本特点：沿指针搜索（或称扫描），它是很多算法的基础，在本书后面的图、树等章节中这一特点会更加明显。

1．建表

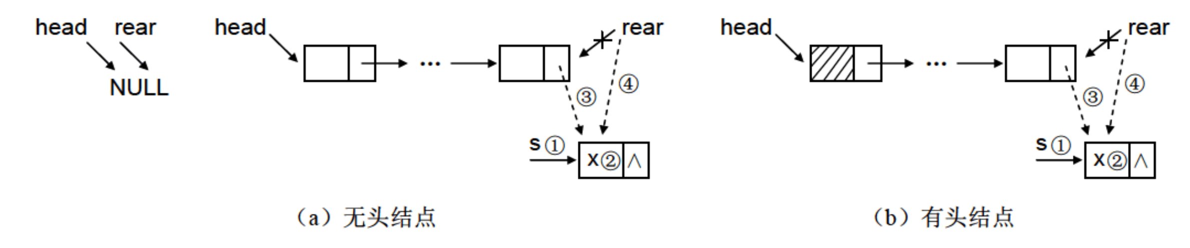
本来使用链表的第一步应该是初始化，但这里为了引入头结点的概念，先介绍单链表的建立。这样一来，初始化的有关工作要在建表时完成。

假设线性表中结点的数据类型是字符（这时前述单链表的类型定义中，把结点数据类型从 int 改为 char），我们逐个输入这些字符型的结点，并以'$'为输入结束标志符。动态地建立单链表的常用方法有如下两种。

（1）尾插法建表

该方法是将新结点插到当前链表的表尾上。即从一个空表开始，每读入一个数据，就生成一个新结点，将数据存放到结点的数据域中，再将新结点插入到当前链表的表尾，如此反复，直至读入结束标志为止。

如果链表只有头指针，则每次插入时都要先从表头开始找到表尾，然后才能真正插入。由于在已有 i 个结点的链表上从表头开始找到表尾的时间复杂度为 i，所以建表的时间复杂度将为 $1+2+\cdots+(n-1)=n(n-1)/2=O(n^2)$，效率很低。为此增加一个尾指针 rear，使其始终指向当前链表的尾结点。尾插过程见图 2.6 所示。



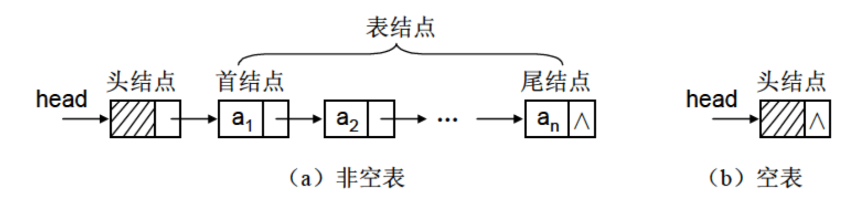
（a）无头结点　　（b）有头结点

图 2.6　尾插法建单链表

当插入第一个结点时，链表由空变为非空，需要修改头指针（使之由空变为指向第一个结点），以后再插入新结点时头指针不变，但要修改尾指针（使之指向新的尾结点）。最后，如果链表不空，要将尾结点的后继指针置空。可见，空表和非空表要分别处理，略显烦琐，具体算法略。

对此问题可采用如下技巧：在链表的第一个结点之前附加一个类型相同的结点，称之为**头结点**，并将头指针指向头结点。这时，无论链表是否为空，头结点总存在，即头指针总不为空。这可使空表和非空表的处理统一起来，而不必单独处理，这就是头结点带来的好处。比如插入第一个结点与以后插入其他结点的操作完全一样，无须特殊处理。

除了头结点外，链表的其他结点统称为**表结点**。表结点中的第一个和最后一个分别称为**首结点**和**尾结点**。带头结点的单链表如图 2.7 所示，图中阴影部分表示头结点的数据域，不存储信息，但在有的应用中，也可用来存放某种特殊标志或表的长度等附加信息。



（a）非空表　　（b）空表

图 2.7　带头结点的单链表

引入头结点后，尾插法建立单链表的算法可简化为：

```
lklist creat2() {         //尾插法建表，有头结点，返回表头指针
  pointer head,rear,s;
  char ch;
  head=new node;          //生成头结点
  rear=head;              //尾指针初值指向头结点
  while(cin>>ch,ch!='$') {  //读入结点值，并检测是否为结束符
    s=new node;           //生成新结点
    s->data=ch;
    rear->next=s;         //新结点插入表尾
    rear=s;               //尾指针 rear 指向新的表尾
  }
  rear->next=NULL;        //尾结点的后继指针为空
  return head;
}
```

注意其中利用 C/C++语言的逗号运算符，将输入结点的语句放到了 while 条件中。

附加头结点后，单链表上的其他很多算法，特别是涉及插入与删除操作时，也可取得类似的效果，即不必单独处理链表是否会由空变为非空，或者由非空变为空这些“边界”

情况，而将空表和非空表统一处理，使有关算法大为简化。另外，一般说来，链表由空变为非空，或者由非空变为空的情况很少，比如只发生一次，大量的时候这类检测实际上是多余的，如果能省掉这些检测语句，还可提高算法的时间性能。所以在实际应用中，经常在单链表上附加头结点。但要注意，头结点逻辑上并不属于线性表，只是一种技术处理。

（2）头插法建表

该方法是将新结点插到当前链表的表头上。如果链表无头结点，头插过程见图 2.8（a），注意这里不必对第一个结点单独处理。如果链表有头结点，则从有头结点的空表开始，每次在头结点后插入新结点，算法如下：

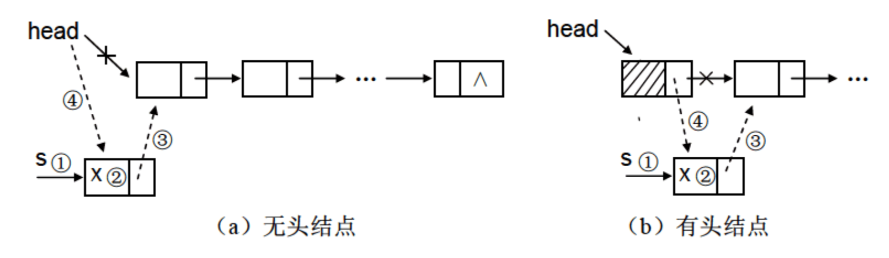


图 2.8 头插法建单链表

```
lklist creat() {          //头插法建表，有头结点，返回表头指针
  lklist head;
  pointer s;
  char ch;
  head=new node;          //生成头结点，头指针初值指向头结点
  head->next=NULL;
  while(cin>>ch,ch!='$') {  //输入结点值，并检测是否为结束符
    s=new node;           //生成新结点
    s->data=ch;           //装入数据
    s->next=head->next;
    head->next=s;         //将新结点插入到头结点后
  }
  return head;
}
```

头插法建立链表的算法也很简单，但链表中结点的次序和输入时相反。若希望二者次序一致，则只能用尾插法建表。

2. 初始化

初始化即建立一个只有头结点的空表，算法如下：

```
lklist initlist() {
  pointer head;
  head=new node;
  head->next=NULL;
  return head;
}
```

3. 求表长

算法思想是从首结点开始沿着指针链向后搜索，逐个统计表结点数，一直统计到尾结点为止，得到的结点总数即表长。由于要沿着链表搜索，需要一个工作变量来指示当前位置，它的变化规律是，每统计完一个结点，就指向下一个结点。求表长的算法如下：

```
int length(lklist head) {
```

```
  int j;
  pointer p;
  j=0;                     //计数器
  p=head->next;            //从首结点开始搜索
  while(p!=NULL) {
    j++;
    p=p->next;             //到下一个结点
  }
  return j;
}
```

4. 按序号查找

在顺序表中，要访问序号为 i 的结点可通过数组下标直接访问，因为数组可随机存取。但链表不行，它的结点在空间上的分布没有规律，不能随机存取，只能从链表开始部分沿着指针链向后顺序查找，一直搜索到第 i 个点为止。这一过程与求表长有些类似，但这里不是一直搜索到尾结点。

设表长为 n，则要查找的结点号 i 应满足 1≤i≤n，但我们有时需要把头结点形式上看成是第 0 号结点（首结点的“前趋”），这样，当 i=0 时返回头结点（比如在后面将介绍的插入算法中，若插入点为第 1 个结点，则需要先找到它的“前趋”，即第 0 号结点）。按序号查找的算法如下：

```
pointer get(lklist head,int i) {//结点号 i 的有效范围是 0≤i≤n
  int j;
  pointer p;
  if(i==0) return head; //0 号结点为头结点
  if(i<0) return NULL;  //位置非法，无此结点
  j=0;                     //计数器
  p=head->next;            //从首结点开始搜索
  while(p!=NULL) {
    j++;if(j==i) break;
    p=p->next;             //没有搜索到第 i 个点，继续下一个结点
  }
  return p;                //未找到时 p 自动为 NULL
}
```

上面 if 语句检查非法位置时，条件本应为 i<0 或 i>n，但链表长度 n 尚未知，故对 i>n 的检测就留给了后面的 while 循环。while 循环结束时有两种可能：找到或未找到第 i 个结点，本应分情况返回 p 或 NULL，但注意到未找到时，p 搜索到尾结点外（即 i>n），p=NULL，返回 p 也即返回 NULL，所以算法返回时未加区分。

5. 定位（按值查找）

算法思想：从首结点开始沿着指针链搜索，逐个检查每个结点的数据域，看它是否是所要查找的结点；若是则返回该结点的地址，否则返回 NULL。算法与按序号查找类似，但查找的目标不同，这里不是找第 i 个点，而是找数据域的值。

```
pointer locate(lklist head,datatype x) {
  pointer p;
  p=head->next;            //从首结点开始搜索
  while(p!=NULL) {
    if(p->data==x) break;
    p=p->next;             //到下一个结点
```

```
  }
  return p;              //未找到时 p 自动为 NULL
}
```

上面 while 循环结束时也没有按找到或没找到分别返回，原因与按序号查找时相同。若定位时要求返回找到的结点号，没找到就返回−1（0 代表头结点），则算法如下：

```
int locate(lklist head,datatype x) {
  int j;
  pointer p;
  j=0;                   //计数器
  p=head->next;          //从首结点开始扫描
  while(p!=NULL) {
    j++;if(p->data==x) break;//找到了 x，退出循环
    p=p->next;           //没找到，继续扫描下一个结点
  }
  if(p!=NULL) return j; //找到了 x
  else return -1;        //没有 x，查找失败
}
```

6．插入运算

实现插入运算 INSERT(L, x, i)的基本步骤如下：

（1）在链表中找到插入位置，这可通过按序号查找（get）来实现。

（2）生成一个以 x 为值的新结点。

（3）将新结点插入。

插入过程见图 2.9，要使插入的结点成为新的第 i 个结点，需要修改原第 i−1 个结点的后继指针，使之指向新结点；新结点的后继是原第 i 个结点。这样，在第（1）步中实际需要找第 i−1 个结点的地址 q 而不是第 i 个结点的地址 p。算法如下：

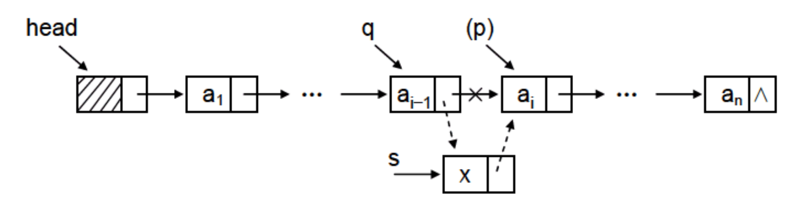


图 2.9　在单链表第 i 个位置插入结点

```
int insert(lklist head,datatype x,int i) {
  pointer q,s;
  q=get(head,i-1);       //找第 i-1 个点
  if(q==NULL) {cout<<"非法插入位置!\n";return 0;}//无第 i-1 个结点，即 i<1 或 i>n+1 时
  s=new node;            //生成新结点
  s->data=x;
  s->next=q->next;       //新结点的后继是原第 i 个点
  q->next=s;             //原第 i-1 个结点的后继是新结点
  return 1;              //插入成功
}
```

注意上面修改指针 s−>next 和 q−>next 的两条语句的顺序不能颠倒，否则若先修改了 q−>next=s；则原第 i 个结点的位置就不知道了。一般规则是先修改新结点的指针（这时不影响其他结点），再修改旧结点的指针。

在链表的插入中还会遇到这种情况：已经知道了某结点的位置（而不是序号），需要在该结点之前或之后插入一个结点。显然，在该结点之前插入一个结点（前插）不方便，需要先从头指针开始找到该结点的前趋（就像上面在第 i 结点前插入一个结点时需要找第 i−1 个结点一样），然后才能真正进行插入，执行时间与插入位置有关，易见其平均时间复杂度为 O(n)；但在该结点之后插入一个结点（后插）就比较方便，可直接进行有关操作，时间复杂度是 O(1)。当找到某结点的前趋后，前插问题实际上便转化成了后插问题。

不找前趋是不是就不能进行前插操作呢？这里给出一个简单的等效方法：先把新结点插在当前结点*p 之后，然后交换这两个结点的内容。这样得到的链表逻辑上与原来在*p 之前插入等效，如图 2.10 所示。主要算法步骤如下：

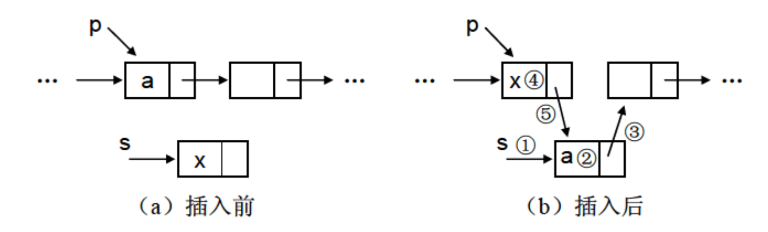


图 2.10 等效的前插操作示意图

```
(1) s=new node;
(2) s->data=p->data;
(3) s->next=p->next;
(4) p->data=x;
(5) p->next=s;
```

显然，改进后的前插算法时间复杂度是 O(1)。注意，由于结点*s 的内容要交换，数据 x 并不需真的装入其中后再交换，而是*p→*s，x→*p；与物理前插相比（这时为 x→*s），多了一次结点数据的复制。若结点数据域的信息量较大，复制会增加一定的时间开销。

比较上述两种插入操作可知，除了在表的第一个位置上的前插操作外，表中其他位置上的前插操作都没有后插操作简便。因此在一般情况下，应尽量把单链表上的插入操作转化为后插操作。

7．删除运算

根据删除运算 DELETE(L, i)的定义，可知其基本步骤为：

（1）找到第 i 个结点，这可通过按序号查找（get）来实现。

（2）删除该结点。

结点删除流程见图 2.11，设当前要删除的结点（第 i 个结点）地址为 p，则删除时要修改其前趋的后继指针，使之指向当前结点的后继。所以在第（1）步，实际上要找第 i−1 个结点的地址 q。算法如下：

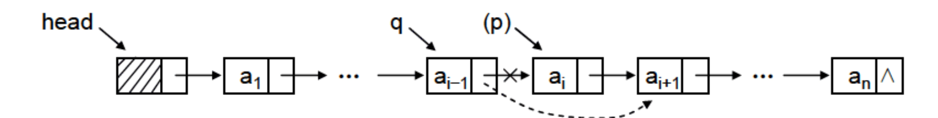


图 2.11 在单链表上删除第 i 个结点

```
int delete(lklist head,int i) {
  pointer p,q;
  q=get(head,i-1);           //找待删结点的直接前趋
  if(q==NULL || q->next==NULL)  //即 i<1 或 i>n 时
    {cout<<"非法删除位置!\n";return 0;}
  p=q->next;                 //保存待删结点的地址，用于释放空间
  q->next=p->next;           //修改前趋的指针
  delete p;                  //释放已删除的结点空间
  return 1;                  //删除成功
}
```

上面的 if 语句有两个条件：q==NULL 表示待删结点的前趋不存在；q->next==NULL 表示待删结点本身不存在。这是因为前趋存在并不能保证待删结点也存在，如链表长度为 n，若要删除第 i=n+1 个结点，它实际上不存在，但它的“前趋”第 i–1 个结点却存在，即终端结点。对不存在的结点实行删除操作，如取它的后继，释放它的空间等是错误的。

注意，上述算法中若没有语句 delete p，则链表上虽然没有了结点*p，但它仍占用内存空间，却又不起作用，形成内存垃圾。随着这种结点的增多，内存浪费越来越严重，可用内存越来越少，最终将会影响到程序的正常运行①。所以链表中不需要的结点要及时将其空间释放。

在链表的删除中还会出现这样一种情况，即已经知道了某结点的位置（而不是该结点的序号），需要删除该结点或其后继。显然，删除该结点本身（自删）不方便，需要先从头指针开始找到该结点的前趋（就像上面删除第 i 个结点时需要找第 i–1 个结点一样），然后才能真正进行删除，执行时间与删除位置有关，易见其平均时间复杂度为 O(n)；但删除该结点的后继（后删）比较方便，可直接进行有关操作，时间复杂度是 O(1)。当找到某结点的前趋后，自删问题实际上便转化成了后删问题。

不找前趋是不是就不能删除当前结点呢？这里给出一个简单的等效方法：先将当前结点*p 后继的值复制到当前结点中，然后删去当前结点的后继，见图 2.12。这样得到的链表逻辑上与原来删除*p 等效。主要算法步骤如下：

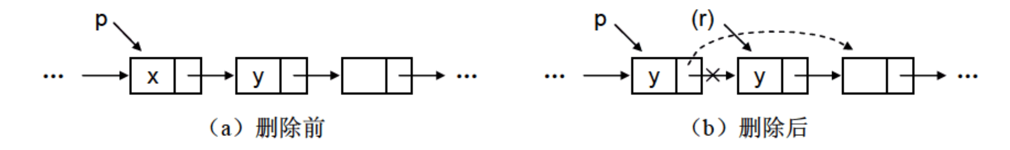


图 2.12　等效的自删操作示意图

（1） r=p->next;
（2） p->data=r->data;
（3） p->next=r->next;
（4） delete r;

但此方法要求*p 有后继，也就是说，它不能是终端结点。与物理自删相比，多了一次结点数据的复制。

---

① 这就是所谓的**内存泄漏**（**memory leak**）。类似，程序结束时最终各链表中的所有结点（包括头结点）也要逐个将其空间释放，这可通过增加销毁函数来完成。

8. 销毁运算

链表的销毁比较简单，就是从头到尾将所有结点空间释放，算法如下：

```
void destroy(lklist head) {
  pointer p,q;
  p=head;           //从头结点开始
  while(p!==NULL) {
    q=p->next;
    delete p;
    p=q;
  }
}
```

以上各算法的时间复杂度除了初始化为 O(1)外，其他都是 O(n)。

从上面的讨论可以看到，链表上实现的插入和删除运算，无须移动结点，仅需修改有关指针。

例 2.4 将单链表中的结点就地逆置（即在原结点上修改，辅助结点空间为 O(1)）。

解：参见图 2.13，设单链表有头结点。运算规则是依次将旧链表中的结点*p 头插到新链表中。头插后*p 的 next 被修改，不能由其搜索到原后继，所以在修改该指针之前，先将其后继*r 的位置保存起来。算法如下：

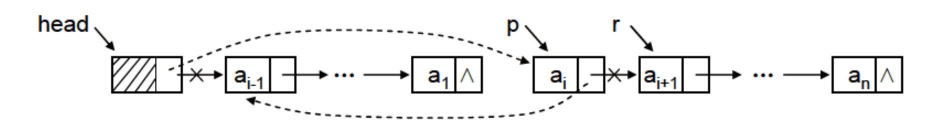


图 2.13 单链表就地逆置（头插法）

```
void reverse(lklist head) {
  pointer p,r;
  p=head->next;             //p 指向旧链表结点
  head->next=NULL;          //新链表从空开始（头结点用旧链表的）
  while(p!=NULL) {
    r=p->next;
    p->next=head->next;     //头插
    head->next=p;
    p=r;
  }
}
```

本例也可直接扫描链表，将各结点的 next 指针改为指向其前趋，具体算法略。

2.3.3 循环链表

循环链表（Circular Linked List）是一种首尾相接的链表。其特点是无须增加存储量，仅对表的链接方式稍作改变，就可使表的处理更加方便灵活。

在单链表中，将尾结点的指针域由 NULL 改为指向头结点或首结点，就得到了单链形式的循环链表，简称为**单循环链表**。这相当于将尾结点的后继视为头结点或首结点，将头结点或首结点的前趋视为尾结点。类似地，还有多重链的循环链表，如双循环链表（见下节）。单循环链表中，表中所有结点被链在一个环上，多重循环链表则是将表中结点链在多个环上。循环链表中头结点也能起到使空表和非空表的处理一致的作用。带头结点的单循

环链表如图 2.14 所示，其中空循环链表只有头结点并自成循环。

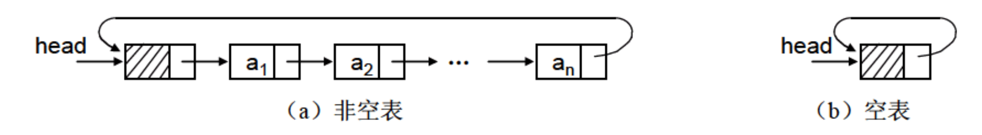


图 2.14 单循环链表示意图

在用头指针表示的单循环链表中，找首结点 a_1 的时间是 O(1)，然而要找到尾结点 a_n，则须从头指针开始遍历整个链表，其时间是 O(n)。在很多实际问题中，表的操作常常要在表的首尾位置上进行，此时头指针表示的单循环链表就显得不够方便。如果改用尾指针 rear 来表示单循环链表，见图 2.15，则查找首结点 a_1 和尾结点 a_n 都很方便，它们的存储位置分别是 rear−>next−>next 和 rear，显然，查找时间都是 O(1)。因此，使用中常用尾指针来表示单循环链表。

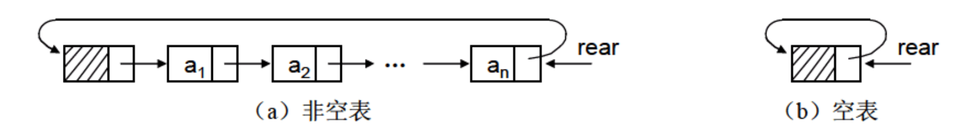


图 2.15 用尾指针表示的单循环链表

在单循环链表中，从任一结点出发都可访问到表中所有结点，这一优点使得某些运算易于实现。而在单链表中，只有从头结点（或首结点）开始才能扫描到表中全部结点，从某一结点出发只能访问到该结点及其后续结点，无法找到该结点之前的其他结点。

例 2.5 多项式的表示。

解：一元 n 次多项式：

$$f(x)=a_nx^n+a_{n-1}x^{n-1}+\cdots+a_1x+a_0$$

由 n+1 个系数唯一确定，在例 1.1 曾提到，这些系数中可能有很多是零，为了节约空间，并与书写习惯一致，我们只存储非零系数。可将各系数组织成一个链表，每个结点表示一个系数项，包含三部分信息：系数值、指数和后继指针，其中后继指针指出下一个系数结点的位置，从而将各个结点链接起来，结点结构的示意图见图 2.16（a）。以多项式 x^4+3x^2+2 为例，它可用图 2.16（b）所示的链表表示（也可以用非循环链表表示）。

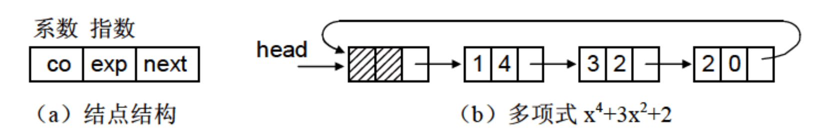


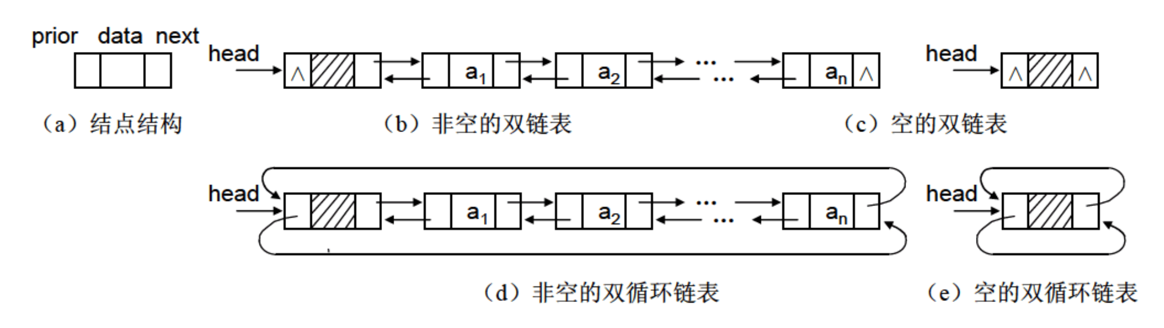
图 2.16 多项式的循环链表表示

结点的类型定义与单链表类似，只是数据域有两个：co 和 exp，这里略。

2.3.4 双链表

在单链表中，每个结点含有后继指针，因此找某个结点的后继比较方便，由后继指针直接可得；但找前趋就不方便了，需要进行查找：如果是单循环链表，可从该结点出发沿后继指针逐个查找，时间耗费是 O(n)；如果不是循环链表，则只能从头结点或首结点出发进行查找，平均时间耗费也是 O(n)。若希望能快速确定一个结点的前趋，则可在单链表的

每个结点里再增加一个指向其前趋的指针域（见图 2.17（a））。这样形成的链表含有指向前趋和后继两个方向的链，称之为**双（向）链表**（Doubly Linked List），见图 2.17（b）、（c）。



（a）结点结构　（b）非空的双链表　（c）空的双链表

（d）非空的双循环链表　（e）空的双循环链表

图 2.17　双链表示意图

双链表类型定义如下：

```
typedef int datatype;              //结点数据类型，假设为 int
typedef struct dnode *dpointer;
struct dnode {
  datatype data;
  dpointer prior,next;
};
typedef dpointer dlklist;
```

和单链表类似，双链表一般也由头指针 head 唯一确定，增加头结点也能使双链表上的某些运算变得方便，将头结点和尾结点链接起来也能构成循环链表，称为**双（向）循环链表**，如图 2.17（d）和图 2.17（e）所示。

回顾单链表的插入和删除运算，其前插不如后插方便，删除某结点*p 自身不如删除*p 的后继方便，原因是表中只有一条后继链，运算中需要查找结点的前趋，不方便。双链表结构是一种对称结构，既有前趋链又有后继链，这就使得两种插入操作和两种删除操作都方便。设指针 p 指向双链表的某一结点，则双链表结构的对称性可用下式刻画：

```
p->prior->next=p=p->next->prior
```

亦即结点*p 的存储位置既存放在其前趋结点*(p->prior)的后继指针域中，也存放在其后继结点*(p->next)的前趋指针域中。

双链表的运算如求表长、按序号查找、定位等与单链表基本相同，主要不同是插入和删除运算，因为这时要修改两条链。下面考虑两个对单链表运算比较困难的情况。

图 2.18　双链表的前插操作

（1）在结点*p 之前插入 x，如图 2.18 所示，主要语句为：

```
① s=new dnode;
② s->data=x;
③ s->next=p;
④ s->prior=p->prior;
⑤ p->prior->next=s;
⑥ p->prior=s;
```

这里要修改 4 个指针。注意其中语句的次序：语句⑥必须位于语句④、⑤之后，否则，先打断了 p–>prior 这条链，以后就不能从 p 获得 p 的前趋了。一般规则与单链表相同，即先修改新结点的指针（这时不影响其他结点），再修改旧结点的指针，但这里涉及两个旧结点，则先修改其原值不再有用的指针。

可类似写出在结点*p 之后插入 x 的操作步骤（略），可以看到，对双链表来说，前插并不比后插困难，它们一样方便。

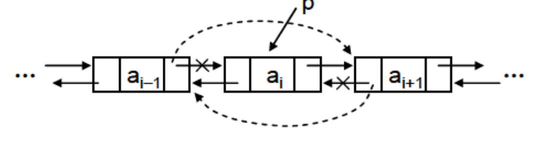


图 2.19　双链表删除结点*p

（2）删除结点*p 自身。如图 2.19 所示，主要语句为：

```
① p->prior->next=p->next;
② p->next->prior=p->prior;
③ delete p;
```

这里修改两个指针的语句次序可交换。

因为双链表上的前插操作和删除某结点*p 自身的操作都很方便，所以在双链表上实现有关插入和删除时，无须转化为后插操作及删去结点后继的操作。例如，在双链表的第 i 个位置上插入或删除，可直接找到表的第 i 个结点*p，然后用上面的方法处理即可，而不必像处理单链表那样，先找到第 i 个结点的前趋才能进行，所以双链表表示显得更为自然和方便。

顺便指出，双链表在实现时，也可只用一个指针域，但该指针域存放的是前趋和后继地址的异或值：prior^next（对头结点：NULL^next=next，对尾结点 prior^NULL=prior）。这样，从首结点开始，已知其前趋 prior（即 head），就可求出后继 next=(prior^next)^prior。类似地，如果已知某点的后继 next，则可求出其前趋 prior=(prior^next)^next。这里利用了异或运算的特点：连续两次异或得原值。显然，这种处理节省了指针空间，但增加了运行时间。

2.3.5　静态链表

以上介绍的各种链表都是由指针实现的，链表中结点空间的分配和回收（即释放）由系统提供的运算符 new 和 delete 动态执行，故称为**动态链表**。但是有的高级语言没有“指针”数据类型，比如 BAS1C、FORTRAN 等，这是否就不能使用链表呢？

在 C/C++语言中，指针指的是数据的物理地址，即其存储位置。注意到这样一个事实，如果一批数据是用数组组织的，则为了指出某个元素的位置，只需指出数组的下标就可以了，这样，数组的下标实际上也是一种“指针”。按此思想，我们就可以用数组来实现链表：将数据存放到数组中，相互间的逻辑关系用数组下标来表示。类似单链表，这里每个结点也要存储两个信息，一个是结点本身的数据，一个是其后继结点的数组下标（指针）。这样，所需数组就是一个结构数组，每个数组元素由两部分组成。最后这种链表也由头指针唯一确定。

例如，图 2.20（a）所示的结构数组中存放的就是线性表(a_1, a_2, a_3, a_4)。对比图 2.4，不难发现，这两种链表本质上并没有什么不同：图 2.4 中的指针是结点的物理地址，这里的

指针是结点的数组下标，它本质上与物理地址是对应的。为了区分，把这里的指针称为“游标（Cursor）”。但在不至于混淆时，习惯上还是把游标称作指针。接下来，也可以写出“分配结点”和“回收结点”的过程。

显然，图 2.20 中空指针应该为−1，这时不能再用 C/C++语言的 NULL 来表示空指针了，因为它被定义为 0。为此，一个方法是将数组的 0 号单元视为“空”而不用，从而继续用 NULL 表示空指针。本节采用另一个方法，即将空指针用另一符号 NIL 表示，并将其定义为−1。

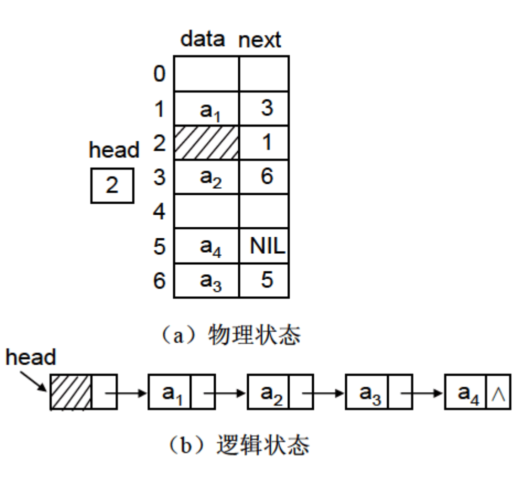


图 2.20 静态链表示意图

可见，用游标实现链表的方法是：定义一个规模较大的结构（有些语言称记录）数组作为备用结点空间（即存储池）。当申请结点时，从存储池内取出一个结点；当释放结点时，将结点归还到存储池内。由于要预先给备用结点静态分配一个连续空间，故把用游标实现的链表称为**静态链表**（Static Linked List）。

静态链表的类型定义如下：

```
const int maxsize=100;      //存储池的容量，假设为 100
typedef int datatype;       //结点数据类型，假定为 int
typedef int cursor;         //游标类型
const cursor NIL=-1;        //静态链表空指针
typedef struct {
  datatype data;            //数据域
  cursor next;              //指针域
} snode;                    //结点类型
snode nodepool[maxsize];    //存储池
cursor sp;                  //游标变量
```

对于双链表，在定义中要增加一个前趋指针。注意，头指针和其他指针的类型是 cursor。如果所用语言没有结构（记录）类型，则可将存储池用两个数组 data[maxsize] 和 next[maxsize]来实现。

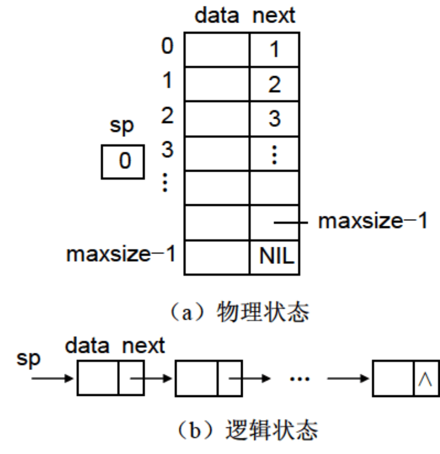


图 2.21 初始可用空间表

为了便于结点空间的分配和回收，通常将存储池中所有的可用结点链成一个链表，并称之为可用空间表或备用空间表，它的头指针设为 sp，类型是 cursor。在结点分配和回收中要使用 sp，为了避免参数的频繁传递，这里将 sp 设为全局量。初始的可用空间表 sp 是数组的全部单元，可简单地将它们依次串接起来，见图 2.21，对应的初始化算法如下：

```
void initialize() {
  int i;
  for(i=0;i<maxsize-1;i++)      //依次链接存储池中各结点
```

```
    nodepool[i].next=i+1;
  nodepool[maxsize-1].next=NIL;//最后一个结点的指针域为 NIL
  sp=0;
}
```

可用空间表 sp 建立后，就可以实现结点空间的分配和回收。为此，我们定义两个函数 newnode 和 deletenode 来实现 new 和 delete 的功能。当需要分配一个结点时调用函数 newnode，从表 sp 上取走第一个结点空间，并在 sp 中删除该结点。若要归还一个结点，则调用函数 deletenode，将该结点插入到表 sp 的头上。下面是实现这两个函数的算法。

```
cursor newnode() {            //从 sp 表上分配一个结点,返回分配到的可用结点
  cursor p;
  if(sp==NIL) return NIL;     //未分配到结点
  p=sp;                       //p 指向分配到的结点
  sp=nodepool[sp].next;       //从 sp 表上删除分配走的结点
  return p;                   //返回分配到的结点
}
deletenode(cursor p) {        //将 p 所指的结点归还到 sp 表中
  nodepool[p].next=sp;
  sp=p;                       //将释放的结点插入表 sp 的第一个结点之前
}
```

在经过多次结点的分配和回收后，存储池的空间就不一定连续而变得比较零乱了，这与动态链表类似，但并不影响我们的使用。

需要指出，静态链表申请结点空间时，一定要检查申请的结果是否为空，若空则意味着存储池已满，可用空间表已用完。这是因为存储池数组大小总是有限的，使用中要考虑空间溢出问题。而在动态链表中，我们一般不必进行这种检查，因为系统分配的用户内存较大，一般不需要考虑空间溢出问题，除非程序或数据量特别大。

定义了 newnode 和 deletenode 之后，并注意到游标和指针的对应关系，就不难由动态链表上的算法得到静态链表上的相应算法。游标和指针的对应关系是：静态链表中的游标 p 是向量的下标，它所指的结点是 nodepool[p]，由 p 向后搜索的语句为“p=nodepool[p].next;”，动态链表中的指针 p 是结点的地址，它所指的结点是*p，由 p 向后搜索的语句为“p=p->next;”。这里仅举两个例子。

例 2.6　已知静态链表中结点的值有正有负，编写算法删除其中值为负的结点。

解：删除过程见图 2.22，其中 p 指向当前结点，q 指向其前趋，算法如下。

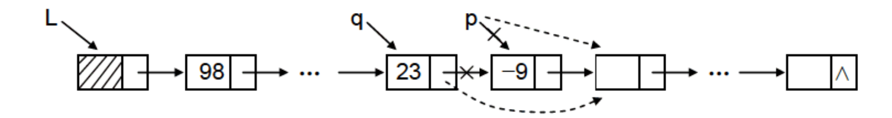


图 2.22　删除静态单链表中的负数结点

```
void deletes(cursor L) {
  cursor p,q;
  q=L;                              //q 从头结点开始（后继 p 从首结点开始）
  while(p=nodepool[q].next,p!=NIL)
    if(nodepool[p].data<0) {        //当前结点为负，删除之
      nodepool[q].next=nodepool[p].next;
      deletenode(p);
    }
```

```
    else {                          // 当前结点非负，向后推进
      q=p;
    }
}
```

其中每次 p 取新值的语句写在了 while 条件中（类似前述单链表的建表算法）。

例 2.7 在带头结点的静态链表上第 i 个结点 a_i 之前插入一个新结点 x。

解：算法与动态链表插入完全类似（参见图 2.9），先找到插入点的前趋，再插入新结点。

```
int insert(cursor L,datatype x,int i) {
  cursor q,s;
  q=get(L,i-1);                     //找 ai 前趋结点(算法略)
  if(q==NIL) {cout<<"插入位置非法\n";return 0;}
  s=newnode();
  if(s==NIL) {cout<<"空间已满，不能插入\n";return -1;}//未分配到结点，出错
  nodepool[s].data=x;               //插入结点 x
  nodepool[s].next=nodepool[q].next;
  nodepool[q].next=s;
  return 1;                         //插入成功
}
```

2.4 顺序表和链表的比较

前面几节介绍了线性表的两种存储结构：顺序表和链表。其中，顺序表是用数组实现的，链表是用指针或游标实现的。用指针实现的链表，结点空间是动态分配的，称之为动态链表；用游标模拟指针实现的链表，结点空间是静态分配的，称之为静态链表。顺序表和链表各有优缺点，在实际使用中如何选择呢？这要根据具体问题的要求和性质来决定，通常有以下几方面的考虑。

1．基于空间的考虑

顺序表的存储空间是静态分配的，在程序执行之前必须明确规定它的存储规模。若线性表的长度 n 变化较大，则存储规模难于预先确定，估计过大将造成空间浪费，估计太小又将使空间溢出机会增多。

静态链表中，初始存储池虽然也是静态分配的，但若同时存在若干个结点类型相同的链表，则它们可以共享空间，使各链表之间能够相互调节余缺，减少溢出的机会。动态链表的存储空间是动态分配的，只要内存空间尚有空闲，就不会产生溢出。

在顺序表中，结点间的逻辑关系由其物理位置反映，即不必用额外的空间来表示逻辑关系。但链表中各结点的物理位置是任意的，每个结点除了保存其自身的数据外，还需要额外的空间（即指针域或游标域）来表示逻辑关系，这从存储密度来讲是不经济的。所谓**存储密度**（Storage Density）是指结点数据本身所占的存储量和整个结点结构所占的存储量之比，即：

存储密度=（结点数据本身所占的存储量）/（结点结构所占的存储总量）

一般地，存储密度越大，存储空间的利用率就越高。显然，顺序表的存储密度为 1，而链表的存储密度小于 1。例如，假设单链表的结点数据为整数，且指针所占的空间和整型量相同，则单链表的存储密度为 50%。这就是说，若不考虑顺序表中的备用结点空间，

则顺序表的存储空间利用率为 100%，而单链表的存储空间利用率为 50%。

因此，当线性表的长度变化较大，难以估计其存储规模时，以采用动态链表作为存储结构为好；当线性表的长度变化不大，易于事先确定其大小时，为了节约存储空间，宜采用顺序表作为存储结构。

2．基于时间的考虑

顺序表由向量实现，是一种随机存取结构，对表中任一结点都可在 O(1)的时间内直接存取；而链表中的结点，一般需从头指针起顺着链扫描才能取得，时间复杂度为 O(n)。

在链表中的任何位置上进行插入和删除，都只需要修改指针。而在顺序表中进行插入和删除，平均要移动表中一半左右的结点，尤其是当每个结点的信息量较大时，移动结点的时间开销就相当可观。

因此，若线性表的操作主要是进行查找，很少做插入和删除操作时，采用顺序表做存储结构为宜；对于频繁进行插入和删除的线性表，宜采用链表做存储结构；若线性表的插入和删除主要发生在表的首尾两端，则采用尾指针表示的单循环链表为宜。

3．基于语言的考虑

一般高级语言都提供了数组类型，所以顺序表的实现比较容易。但很多高级语言并没有提供指针类型，这时若要采用链表结构，则需使用用游标实现的静态链表。虽然静态链表在存储分配上有不足之处，但它和动态链表一样，具有插入和删除方便的特点。

值得指出的是：即使是对那些具有指针类型的语言，静态链表也有其用武之地。特别是当线性表的长度不变，仅需改变结点间的相对关系时，静态链表可能比动态链表更方便。

习　题　二

2.1　{a, b, c, d, e, f}是线性表吗？

2.2　试述头指针、头结点、首结点的区别，并说明头指针和头结点的作用。

2.3　循环链表的主要优点是什么？

2.4　试对如下情况，写出顺序表的类型定义和有关算法：

（1）假设数组空间在运行时动态分配。

（2）假设数组空间从下标 1 开始使用。

2.5　试设计算法，删除顺序表中值大于 low 且小于 high 的结点（若存在这样的结点）：

（1）顺序表无序。

（2）顺序表递增有序。

2.6　试设计算法：

（1）删除顺序表中重复的元素（相同的元素只保留第一个），要求元素的移动次数要少。

（2）删除顺序表中重复的零元素，要求元素的移动次数要少。

（3）删除顺序表中所有的零元素，要求元素的移动次数要少。

2.7　试设计算法，仅用一个辅助结点，完成下列要求，并分析算法的时间复杂度：

（1）将数组中的元素循环右移 k 位。

（2）将数组[a_1, a_2, …, a_n, b_1, b_2, …, b_m]调整为[b_1, b_2, …, b_m, a_1, a_2, …, a_n]。

2.8 假定用递增有序的线性表表示集合，编写算法求集合的交集。

2.9 删除单链表中结点的值在 low 和 high 之间的结点（若存在这样的结点）。

2.10 试设计算法，将两个递增有序的单链表就地合并为一个递增有序的单链表。

2.11 试设计算法，判断两个单链表的结点是否构成如下数列：

（1）递增　　（2）等差　　（3）等比

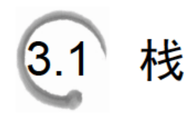
2.12 试写出两个一元多项式相加的算法。

第3章 栈、队列和串

栈和队列是两种特殊的线性表，它们的逻辑结构和线性表相同，只是其运算规则较线性表有一些限制，故又称为运算受限的线性表。很多问题可直接用栈和队列来描述，有些算法也必须借助它们来实现，栈和队列在程序设计中有着广泛的应用。

串（又称字符串）也是一种特殊的线性表，只是其元素的类型有限制，即每个结点仅由一个字符组成。事物的名称、源程序等都是串。计算机上非数值处理的对象也基本上都是串。

本章将介绍栈、队列和串的有关概念、存储结构，以及基本运算。



3.1 栈

3.1.1 栈的基本概念

栈（Stack）是限制仅在表的一端进行插入和删除运算的线性表，通常称插入、删除的这一端为**栈顶**（Top），另一端称为**栈底**（Bottom）。插入和删除也分别称为进栈（入栈）和出栈（退栈）。当表中没有任何元素时称为空栈。处于栈顶位置的元素称为栈顶元素。

根据上述定义，每次删除（退栈）的总是当前栈中“最新”的元素，即最后插入（进栈）的元素，而最先插入的元素被放在栈的底部，要到最后才能删除。在图 3.1 所示的栈中，元素是以 $a_1, a_2, \cdots, a_n$ 的顺序进栈，而退栈的次序却是 $a_n, a_{n-1}, \cdots, a_1$。也就是说，栈的修改是按“后进先出”的原则进行的。因此，栈又称为**后进先出**（LIFO，Last In First Out）或**先进后出**（FILO，First In Last Out）的线性表。

出栈 进栈

栈顶→ a_n

⋮

a_2

栈底→ a_1

图 3.1 栈示意图

栈的例子在日常生活中也可见到，如一叠书或一叠盘子，若规定从中取出一件或放入一件都只能在顶部进行，那它就是一个栈。又如一台机器的拆、装过程显然应该按栈的方式进行。

栈在计算机系统中的应用更多，如记录中断返回地址（断点）的结构就是栈。在中断发生时，为了处理完中断事件后恢复被中断事件的继续执行，需记下断点。在允许多级中断的情况下，中断处理过程又可能被其他中断所中断，因此，系统可能要保存多个断点。由于中断恢复是先恢复最近被打断的过程，于是要求断点的取出次序与保存次序相反，即后保存的先取出，这正是栈的特征。

设 S 表示栈，栈的基本运算有 5 种：

（1）初始化 INITIATE(S)。加工型运算，其作用是设置一个空栈。此后其他操作才能进行。

（2）判栈空 EMPTY(S)。引用型运算，若栈 S 为空，返回 1，否则返回 0。

（3）进栈 PUSH(S, x)。加工型运算，在栈顶插入（压入）元素 x，x 成为新的栈顶元素。

（4）退栈 POP(S)：加工型运算，将栈顶元素删除（弹出），并返回该元素。

（5）取栈顶 GETTOP(S)：引用型运算，取栈顶元素，但不删除它。这点不同于 POP(S)。

根据需要，还可增加判栈满 FULL(S)的运算。以上运算（3）～（5），对特殊情况（如栈满或栈空时）需要特殊处理，如返回特殊标志等。

由于栈是运算受限的线性表，因此线性表的存储结构对栈也适用，即一般采用顺序存储和链式存储。栈顶位置随着进栈和退栈操作而变化，故需要一个变量 top 来指示它的当前位置。通常称 top 为栈顶指针。栈底位置不变，且运算也不涉及它，故不需要设置栈底指针。

3.1.2 栈的顺序实现

栈的顺序存储结构简称为**顺序栈**，它是运算受限的顺序表。顺序栈用向量来实现。向量两端的位置是固定不变的，可将栈底设为其中的任何一端，但习惯上设在下标小的那端；为指出栈顶位置，实际上只需指出它在数组中的下标即可，故栈顶指针 top 取为整型。由于插入和删除只能在栈顶进行，故顺序栈中的插入和删除操作不存在元素移动的问题。顺序栈的类型定义如下：

```
typedef int datatype;   //栈元素的数据类型，假设为整型
const int maxsize=100;  //栈的容量，元素最多不能超过它，此处设为 100
typedef struct {
  datatype data[maxsize];
  int top;
} sqstack;              //顺序栈类型定义
```

顺序栈被定义为一个结构类型，它有两个域：data 和 top。data 为一维数组，用于存放栈的元素，datatype 为栈元素的类型，需要根据实际情况来指定，上面设为 int。top 即栈顶指针（指向真正的栈顶①），它的有效范围为 0～maxsize−1。如果栈底位置固定在向量的低端，那么 top=−1 表示栈空；top=maxsize−1 表示栈满。

与顺序表类似，顺序栈也可从数组下标 1 开始使用，数组大小定义为 data[maxsize+1]，data[0]不用，或用来表示栈顶指针。这时 top=0 表示栈空；top=maxsize 表示栈满。

当栈满时再做进栈运算则产生空间溢出，简称“**上溢**”；当栈空时再做退栈运算也产生溢出，简称“**下溢**”。上溢是一种出错状态，应该设法避免；下溢则可能是正常现象，因为在程序中使用时，栈初态或终态都是空栈，所以下溢常常用来作为程序控制转移的条件。

图 3.2 说明了在顺序栈中做进栈和退栈运算时，栈中元素和栈顶指针的关系。注意：退栈后，元素 D、C 并没有必要擦去，但已不起作用，以后有元素进栈时，自动将其覆盖。

---

① 有的文献中 top 指向真正的栈顶的后一个位置，即 top−1 才是真正的栈顶。这时 top=0 表示栈空；top=maxsize 表示栈满；相应地，栈运算的有关实现需略作调整。

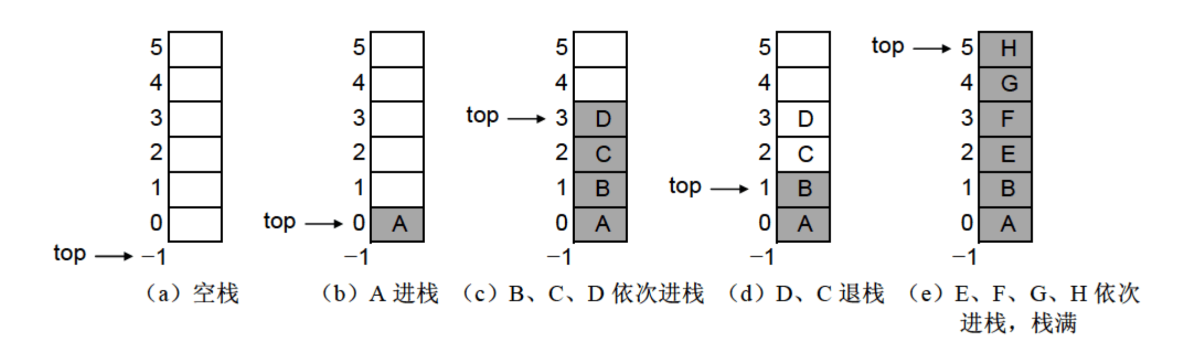


图 3.2　栈顶指针和栈中元素之间的关系

在顺序栈上实现栈的 5 种基本运算如下。

1. 初始化

```
void init_sqstack(sqstack *sq) {
  sq->top=-1;
}
```

2. 判栈空

```
int empty_sqstack(sqstack *sq) {
  if(sq->top==-1) return 1;
  else return 0;
}
```

3. 进栈

```
int push_sqstack(sqstack *sq,datatype x) {
  if(sq->top==maxsize-1) {cout<<"栈满，不能进栈！\n";return 0;}  //上溢
  sq->data[++sq->top]=x;    //栈顶指针加 1，将 x 插入当前栈顶
  return 1;
}
```

4. 退栈

```
int pop_sqstack(sqstack *sq,datatype *x) {      //栈顶元素值由参数返回
  if(sq->top==-1) {cout<<"栈空，不能退栈！\n";return 0;} //下溢
  *x=sq->data[sq->top--];    //取出栈顶元素值给 x，栈顶指针减 1
  return 1;
}
```

5. 取栈顶

```
int gettop_sqstack(sqstack *sq,datatype *x) {  //栈顶元素值由参数返回
  if(sq->top==-1) {cout<<"栈空，无栈顶可取！\n";return 0;}//栈空
  *x=sq->data[sq->top];      //取出栈顶元素值给 x
  return 1;
}
```

注意，以上退栈、取栈顶时，栈顶元素值不是通过函数值来返回的，因为当栈空时无返回对象，需要特殊处理（如返回某个特定的“空值”）。另外，前已指出，退栈时，原栈顶元素并没有必要真正擦去，所以在算法 pop_sqstack 中，删去栈顶元素只是将栈顶指针减 1，即该元素在下次进栈之前仍然是存在的，见图 3.2（d）。

当一个程序中同时使用多个顺序栈时，为了防止上溢，需要为每个栈分配一个较大的空间，但实际上某一栈发生上溢时，可能其余栈剩余的空间还很多，如果我们将这多个栈安排在同一个向量里，即让多个栈共享存储空间，就可以相互调节余缺，这样既可节约存储空间，又可降低发生上溢的概率。当然，这样做的一个前提是，这些栈的类型要相同；还要求各个栈不会同时出现最多元素的情况。

以同时使用两个栈为例，将两个栈的栈底分别设在向量空间的两端，使两个栈各自向中间延伸，见图 3.3。这样当一个栈的元素较多，超过向量空间的一半时，只要另一个栈的元素不多，那么前者就可以占用后者的部分存储空间，只有当整个向量空间被两个栈占满（即两个栈顶相遇）时，才发生上溢。因此，两个栈共享一个长度为 m 的向量空间和两个栈分别占用两个长度为 m/2 的向量空间相比，前者发生上溢的概率比后者要小得多。特别地，如果已知这 2 个栈在任何时刻的总长度都不会超过 m，则该方法是非常实用的。

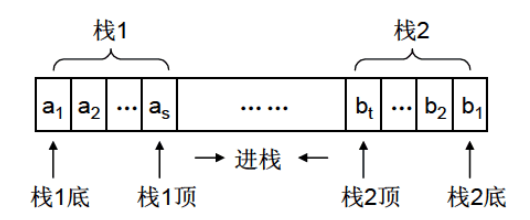


图 3.3　两个栈共享向量空间示意图

但当 k（k>2）个栈共享向量空间时，若某个栈上溢而其余栈中尚有剩余空间，则必须移动某个或几个栈才能为产生上溢的栈腾出空间，处理较烦琐且效率较低。

3.1.3　栈的链接实现

栈的链式存储结构称为**链栈**，它是运算受限的单链表，插入和删除操作限制在链表的一端进行。链表的建立有头插法和尾插法，显然，采用头插法并取链表的头部作栈顶是合适的：在该处插入和删除都很方便，且不必设置头结点（增加头结点并没有什么作用）。栈顶指针 top 就是链表的头指针，链栈由栈顶指针唯一确定，当 top=NULL 时为空栈。

栈中的插入和删除操作不存在元素移动的问题，采用链式存储结构，主要是避免顺序存储中存储区的预申请问题，或者说为了动态利用存储空间。链栈的示意图见图 3.4。

链栈的类型及变量说明与单链表的类似：

```
typedef struct node * pointer;  //结点指针类型
struct node {
  datatype data;
  pointer next;
};                              //链栈结点类型
typedef struct {
  pointer top;
} lkstack;                      //链栈类型
```

data next
top
栈顶
⋮
∧
栈底

图 3.4　链栈的示意图

这里将链栈定义为一个结构类型（但只有一个成员，即栈顶指针 top），一方面可与后面将要介绍的链队列在形式上一致，便于链栈和链队列的对比；另一方面，也可使链栈和

顺序栈有关算法的函数原型基本一致，方便使用。

链栈中的结点是动态产生的，在用户内存空间范围内一般不必考虑上溢问题。下面给出链栈上的 5 种基本运算。

1. 初始化

```
void init_lkstack(lkstack *ls) {
  ls->top=NULL;
}
```

2. 判栈空

```
int empty_lkstack(lkstack *ls) {
  if(ls->top==NULL) return 1;
  else return 0;
}
```

3. 进栈

```
void push_lkstack(lkstack *ls,datatype x) {
  pointer p;
  p=new node;              //申请新结点*p
  p->data=x;;              //新结点 data 域装入 x 的值
  p->next=ls->top;         //改新结点 next 指针，原栈顶成为新结点的后继
  ls->top=p;               //改栈顶指针，新结点成为新的栈顶
}
```

4. 退栈

```
int pop_lkstack(lkstack *ls,datatype *x) {      //栈顶元素值由参数返回
  pointer p;
  if(ls->top==NULL) {cout<<"栈空，不能退栈！\n";return 0;}    //下溢
  p=ls->top;               //保存栈顶结点地址
  *x=p->data;              //取出栈顶元素值给 x
  ls->top=p->next;         //改栈顶指针，原栈顶的后继成为新的栈顶
  delete p;                //释放原栈顶元素空间
  return 1;
}
```

5. 取栈顶

```
int gettop_lkstack(lkstack *ls,datatype *x) {  //栈顶元素值由参数返回
  if(ls->top==NULL) {cout<<"栈空，无栈顶可取！\n";return 0;} //栈空
  *x=ls->top->data;        //取出栈顶元素值给 x
  return 1;
}
```

上述链栈是用动态链表实现的，多个栈的空间自然是共享的。但链栈还可在静态链表上实现，只要插入和删除运算是在静态链表的表头上进行就可，并且也可方便地实现两个以上的静态链栈共享一个向量空间，这比多个顺序栈共享向量空间容易，这里不详细讨论了。

3.1.4　栈的应用举例

栈的应用非常广，只要问题满足 LIFO 原则，均可使用栈做数据结构，本节仅举几例，

在以后的章节中还会经常借助栈来解决各种问题。

例 3.1 编写一个简单程序，从终端上接收字符并可进行一定的编辑。

解：假定在字符接收过程中需要以下功能：

（1）退格，即删除前面的一个字符，用字符“#”代表。

（2）作废，即删除前面的所有字符，用字符“@”表示。

（3）结束，即结束本次输入，用字符“$”表示。例如，假设从键盘上输入串“You@How a#are You!$”，则实际上表示“How are You!”。

根据题意，可用栈来实现这个要求：若读入的字符是“#”，则退栈；若读入的字符是“@”，则退栈到空；若读入的字符是“$”，则编辑结束；当读入其余字符时，则进栈。具体算法如下：

```
sqstack sq;                //顺序栈 sq 设为全程量
void edit() {              //编辑好的字符串在 sq 中
  char ch,x;
  init_sqstack(&sq);       //初始化顺序栈 sq
  while(cin>>ch,ch!='$') {  //读入一个字符，若不是结束符则循环
    switch(ch) {
      case '#': pop_sqstack(&sq,&x);break;
      case '@': while(!empty_sqstack(&sq))
                  pop_sqstack(&sq,&x); break;    //清空栈
      default: push_sqstack(&sq,ch);
    }
  }
}
```

例 3.2 设有 A、B、C、D 四辆车依次进入一个栈式结构的车库，问从库中倒出来的车可不可能是 A、D、B、C 的顺序。

解：这里规定了入库即进栈次序，并不意味着已入栈的车中途不能出栈，也就是说进栈、出栈可交替进行。为判断出栈顺序的可能性，一个简单的方法是用图形来模拟，即先画一个空栈，然后根据进出栈序列的要求来模拟相应的进出栈过程。对目标序列{A, D, B, C}：

（1）第一个出栈的是 A，这可通过 A 入栈，马上出栈得到。

（2）第二个出栈的是 D，这可在（1）基础上 B 入栈、C 入栈、D 入栈、D 出栈得到。

（3）第三个要让 B 出栈，这不可能了，因为第（2）步后栈内有 B 和 C，但 C 为栈顶，只有 C 出栈了 B 才可能出栈。所以{A, D, B, C}不是可能的出栈序列。

如果涉及的序列元素较多，用上面的方法来模拟就比较烦琐了。实际上，这类问题是有规律可循的：设 n 个元素按大小依次入栈，则在可能的出栈序列中，对任一元素 k，其后面的元素或者都大于 k；或者小于 k 的元素递减排列（参见习题 3.2）。对本例{A, D, B, C}，因 D 后面比它小的{B, C}不是递减排列，故不是可能的出栈序列；又如{A, C, B, D}则是可能的出栈序列，相应的进出栈过程为：A 进→A 出→B 进→C 进→C 出→B 出→D 进→D 出。

与此例相关的另一类问题是，当入栈次序一定时，可能的出栈序列有多少。显然，n 个数据的排列有 n！种，但未必都是可能的出栈顺序。当元素很少时可逐个检查其排列，如 3 个元素{A, B, C}的排列有 3！=6 种：ABC、ACB、BAC、BCA、CAB、CBA，检查发现 CAB 不可能，故可能的出栈序列有 5 种。当元素较多时，逐个检查其排列就不合适了。

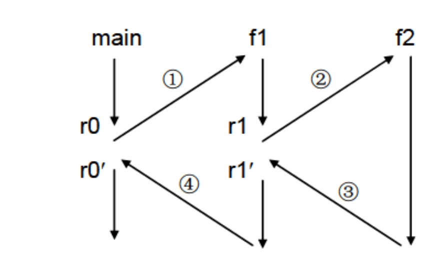
一般地，n 个元素在入栈顺序一定时，可能的出栈序列数为 $\frac{1}{n+1}C_{2n}^{n}$ （参见习题 3.2）。

例 3.3　用栈实现函数调用。

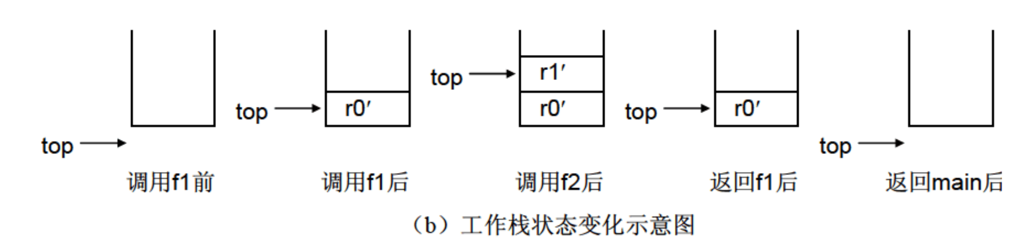
解：考虑如下程序：

```
void main() {          void f1() {          void f2() {
  ⋮                      ⋮                    ⋮
r0:f1();               r1:f2();             }
r0':                   r1':
  ⋮                      ⋮
}                      }
```

该程序有 3 个函数，它们之间存在如下调用关系：主函数 main 在其函数体内 r0 处调用函数 f1，而 f1 又在其函数体内 r1 处调用函数 f2，f2 不调用其他函数。这样，当函数 main 执行到 r0 处时，要调用 f1，于是 main 被“挂起”，f1 开始执行。只有当 f1 执行完后，才返回 main 的 r0′处继续执行剩下的部分。但函数 f1 执行到 r1 处时，又要调用 f2，结果 f1 又被“挂起”，一直等到 f2 执行完后，才返回 r1′处继续执行后继语句。这三个函数的调用与返回关系及运行次序如图 3.5(a)所示。



（a）多级函数调用与返回关系及运行次序



（b）工作栈状态变化示意图

图 3.5　函数的调用与返回次序及相应工作栈状态变化示例

以主调函数 main 为例，在被调用函数 f1 执行完后，需要回到原调用语句的下一条语句处 r0′才能继续执行后面的语句。这就需要在调用转移前保存返回地址 r0′。在多级调用中，就要保存多个返回地址，这样，当某个调用要返回时，应该返回到哪个地址呢？从图 3.5（a）不难看到，在多级调用时，应该返回到最后保存的地址。比如调用从 f2 返回时，已保存的返回地址有两个：r0′和 r1′，正确的是返回到 r1′，即应取出地址 r1′，而它是后保存的。这样，后保存的地址先返回，先保存的地址后返回，正好可用栈来完成。

实际上，用栈来实现函数的调用与返回机制，仅在用低级语言如汇编语言编程时才会考虑；当用高级语言编程时，有关的控制工作由编译系统自动完成了，用户不需考虑。所用的栈称为工作栈。每次函数调用时，将返回地址、参数入栈；另外，局部变量、返回值等也在栈中分配空间，所有这些信息共同构成一个栈顶元素，相当于当前工作环境（常称

活动记录）。栈在函数的调用和返回中，相当于按后进先出的原则切换工作环境（并传递信息）。上面的三个函数执行时，工作栈状态变化示意图如图 3.5（b）所示。

例 3.4 用栈实现递归函数。

解：递归是一个重要的概念，它可用于描述事物，也是一种重要的程序设计方法。简单地说，如果在一个函数或数据结构的定义中直接或间接地应用了其自身（作为定义项之一），则这个函数或数据结构就称为递归定义的，简称**递归**的。递归函数也称为自调用函数。

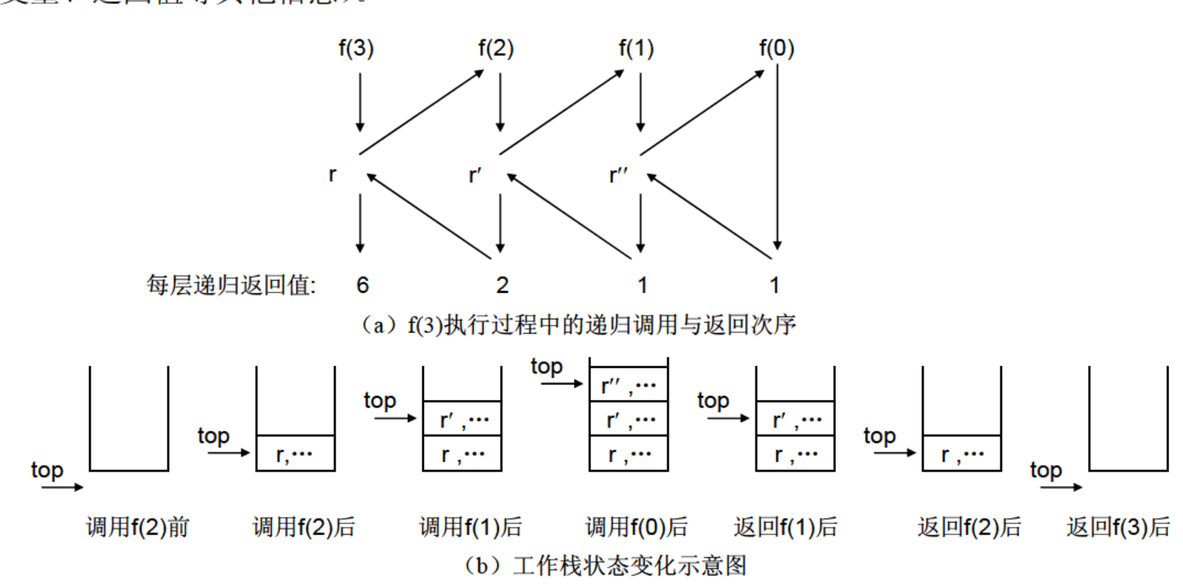
例如，阶乘函数可递归定义如下：

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

由该定义很容易写出相应的递归算法：

```
long f(int n) {
  if(n==0) return 1;
  else return n*f(n-1);
}
```

递归函数的运行引起递归调用。例如，对上面求阶乘的函数，以 n=3 为例，f(3)的执行中出现的递归调用和返回过程如图 3.6（a）所示。显然，递归调用和返回的控制与非递归函数并无本质区别（只是每次调用的是其自己而已），同样由一个工作栈来实现。图 3.6（b）所示为与图 3.6（a）相应的工作栈状态变化过程（栈中除了返回地址外，还有参数、局部变量、返回值等其他信息）。



（a）f(3)执行过程中的递归调用与返回次序

（b）工作栈状态变化示意图

图 3.6 递归函数的调用与返回次序及相应工作栈状态变化示例

关于递归，要特别注意，递归定义不是“循环定义”，它必须满足两个条件：

（1）递归过程中每一次应用自己时，对应的“尺度”要比当前小。

（2）至少存在一个最小的“尺度”，该处的定义不是递归的，从而结束递归。

例如，上面在定义阶乘 n!时应用了阶乘(n−1)!，后者的“尺度”n−1 比当前的“尺度”n 小。另外，阶乘 n! 在最小“尺度”即 0 上的定义不是递归的，直接定义为 1。

上述两个条件实际上构成了递归程序设计的基本原则。在相应的递归程序中，与上面

（1）对应的部分一般称作**递归体**（或称**递归项**），与（2）对应的部分一般称作**递归出口**（或称**终止项**），即递归过程的终止条件，通常写在递归函数的开头。

递归是程序设计中的一个强有力的工具。这是因为：

其一，很多实际问题本身就是递归定义的，这时用递归来实现相应的算法既容易又自然，如上面阶乘的计算、本书后面将介绍的图的深度优先遍历等。

其二，有的数据结构本身就有递归特性，如以后将介绍的二叉树、广义表等，这时对它们的运算用递归描述很方便，而相应的算法用递归实现也很方便。

其三，有些问题虽然没有明显的递归特性，但可很容易地分解（Divide）出若干类似的子问题，分别求解（Conquer）后将结果组合（Combine），就得原问题的解。这就是所谓的“**分治法**”（Divide and Conquer），这类算法用递归实现比较容易，如本书后面将介绍的快速排序、归并排序等。特别地，有些问题如果不用递归求解将非常困难，如著名的 Hanoi 塔问题。

3.2　队列

3.2.1　队列的概念及运算

队列（Queue）也是一种运算受限的线性表。它只允许在表的一端进行插入，而在另一端进行删除。允许删除的一端称为**队头**（Front），允许插入的一端称为**队尾**（Rear）。插入与删除分别称为入队与出队。

队列的特点同现实生活中购物排队过程类似：新来的成员总是加入到队尾（不允许中间插队），每次离开的成员总是队头（不允许中途离队），即当前“最老的”成员离队。换言之，先进入队列的成员总是先离开队列，后进入队列的成员总是后离开队列。因此队列亦称作**先进先出**（FIFO，First In First Out）或**后进后出**（LILO，Last In Last Out）的线性表。队列中的元素出队次序与进队次序相同。

当队列中没有任何元素时称为空队列。在空队列中依次加入元素 $a_1, a_2, \cdots, a_n$ 之后，a_1 是队头元素，a_n 是队尾元素。显然退出队列的次序也只能是 $a_1, a_2, \cdots, a_n$，也就是说队列的修改是按先进先出的原则进行的。图 3.7 是队列的示意图。

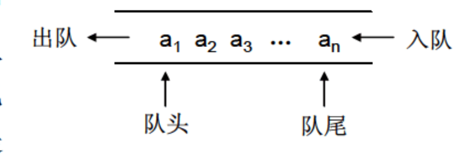


图 3.7　队列示意图

队列的基本运算有以下 5 种：

（1）初始化 INITIATE(Q)。加工型运算，设置一个空队列。

（2）判队空 EMPTY(Q)。引用型运算，若 Q 为空，则返回 1，否则返回 0。

（3）入队 ENQUEUE(Q, x)。加工型运算，将元素 x 插入队列 Q 的尾部。

（4）出队 DEQUEUE(Q)。加工型运算，将队头元素删除，并返回该元素。

（5）读队头 GETHEAD(Q)。引用型运算，取队头元素，但不删除它。

根据需要，还可增加判队满 FULL(Q)的运算。以上运算（3）～（5），对特殊情况（如队列空时），需要特殊处理，如返回特殊标志等。

由于队列是运算受限的线性表，因此线性表的存储结构对队列也适用，即一般采用顺序存储和链式存储结构。队头和队尾位置随着出队和入队操作而变化，故一般需要两个变量 front 和 rear 来指示它们的当前位置。通常称 front 为队头指针，rear 为队尾指针。

3.2.2 队列的顺序实现

队列的顺序存储结构称为**顺序队列**。顺序队列实际上是运算受限的顺序表，它用一个向量空间来存放当前队列中的元素。队头指针和队尾指针分别指出当前队头元素和队尾元素在向量空间中的位置。顺序队列的类型说明如下：

```
const int maxsize=100; //队列的容量，元素最多不能超过它
typedef struct {
  datatype data[maxsize];
  int front,rear;
} sqqueue;              //顺序队列类型
```

通常将尾指针 rear 指向当前队尾元素的真正位置，而头指针 front 指向当前队头元素的前一个位置[①]（不让头、尾指针都指向真正位置是为了某些运算的方便，但不是唯一的方法）。刚开始时，队列的头、尾指针都指向向量空间下界的前一个位置，在此设为−1。以后在出队或入队时，头指针或尾指针值递增变化。图 3.8 说明了在顺序队列中进行入队和出队运算时队列中的元素及其头尾指针的变化情况。注意，与顺序栈类似，出队后的元素并没有真正擦去，但已不起作用。

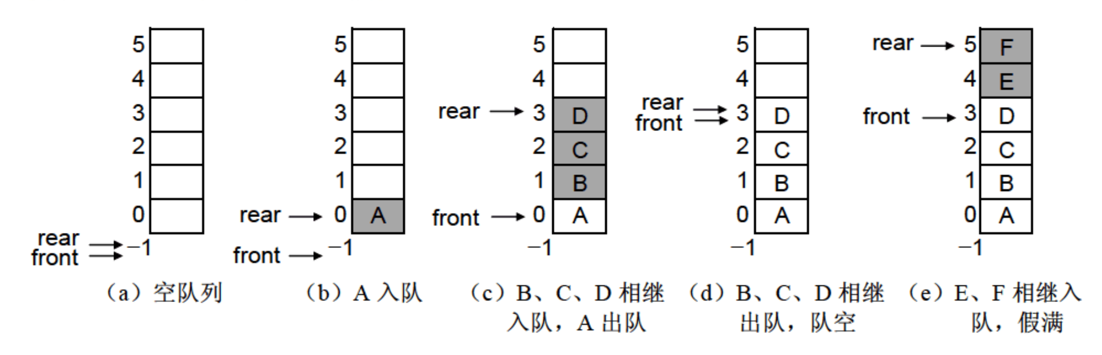


图 3.8 顺序队列运算时的头、尾指针变化情况

与顺序表类似，顺序队列也可从数组下标 1 开始使用，数组大小定义为 data[maxsize+1]，data[0]不用。这时队列开始时，队列的头、尾指针设为 0。

显然，当前队列中的元素个数（即队列的长度）是 rear−front。若 rear=front，则队列长度为 0，即当前队列是空队列，如图 3.8(a)和图 3.8(d)均表示空队列。队列为空时再做出队操作便会产生“下溢”。队满的条件是当前队列长度等于向量空间的大小，即 rear−front=maxsize。队满时再做入队操作会产生“上溢”。

但是，有一种情况，见图 3.8(e)，队列并不满，但尾指针已到数组的上界（rear=maxsize−1），也不能入队了，因为再入队就会溢出到数组空间外。这种现象称为“假

① 或将尾指针 rear 指向当前队尾元素的后一个位置，而头指针 front 指向当前队头元素的真正位置；相应地，队列运算的有关实现需略作调整。

上溢”或“假溢出”。显然，如果中途有出队，则入队 maxsize 次后再入队，必定会假上溢。产生这种现象的原因是，队尾总在队头的后部，随着出、入队的进行，队列整体上向后移动，已出队的原队头空间不能再使用，形成浪费。为克服这一缺点，可以在每次出队时将整个队列向前移动一个位置，或者在发生假溢出时，将整个队列一次性地移动到数组的开始处。显然，这两种方法都会引起大量元素的移动，效果不好。

通常采取的方法是：设想向量 data[maxsize]是一个首尾相接的圆环，即 data[0]接在 data[maxsize−1]之后。我们将这种意义下的向量称为**循环向量**，并将循环向量中的队列称为**循环队列**（Circular Queue），如图 3.9 所示。这时，若当前尾指针等于向量的上界，则再做入队操作时，令尾指针折回到向量的下界，这样就能重复利用已被删除的元素空间了，从而克服了假上溢。

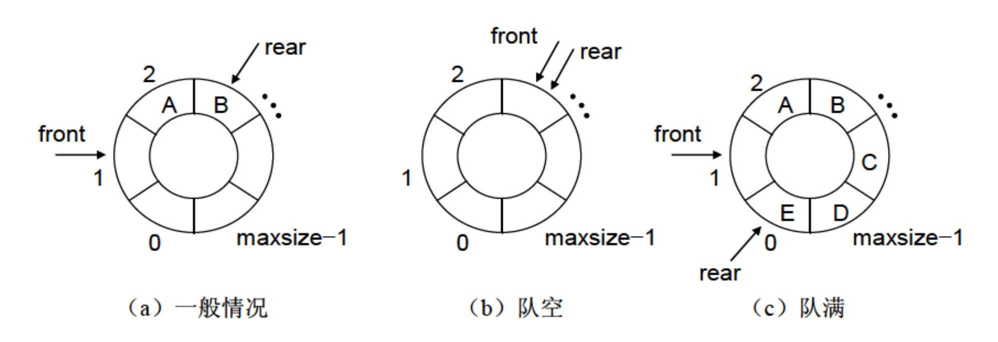


图 3.9　循环队列示意图

在循环队列中做入队操作时，尾指针的变化为：

```
rear=rear+1;if(rear==maxsize) rear=0;
```

如果利用“模运算”，上述语句可简洁地表示为：

```
rear=(rear+1)%maxsize;
```

同样，在循环队列中做出队操作时，头指针的变化为：

```
front=(front+1)%maxsize;
```

为了称呼方便，以上头、尾指针的变化简称为循环意义下的加 1。

在循环队列中，队尾就不一定总在队头的后面了：随着入队的进行，它可能从数组的高端折回到数组的低端，从而位于队头的前面。这样，入队过程中尾指针就可能赶上头指针，即 rear=front，此时队满。同样，出队过程中，头指针也可能赶上尾指针，即 rear=front，此时队空。于是，仅凭等式 front=rear 就无法区分循环队列是空还是满。

对此，一般有以下几种解决方法：

（1）设置长度计数器 n，入队时 n 加 1，出队时 n 减 1，如果 n 为 0 则队空，如果 n 为 maxsize 则队满。

（2）设置标志位 flag，比如，若入队后出现 rear=front，则置 flag=1，其他情况 flag=0。于是队满条件为 flag=1，队空条件为 rear=front 且 flag=0。

（3）空置一单元法：当循环向量中只剩一个元素空间，即有 maxsize 个分量的循环向量已存放 maxsize−1 个元素时，就认为队列已满。这时，尾指针在循环意义下加 1 后等于

头指针，即队满的条件是：

```
(rear+1)%maxsize=front
```

这样，rear=front 就只是队空的条件。队满时的空单元就是 front 所指单元（实际上 front 所指单元总为空）。

显然，前两种方法会增加出、入队的时间开销（且需增加一个辅助空间）。本节采用的是最后一种方法。

和栈类似，在程序中使用的队列，其初态和终态均可为空，因此队空也可作为程序转移的逻辑条件，而队满一般是一个错误状态，应设法避免。

在循环队列上实现的 5 种基本运算如下：

1．初始化

```
void init_sqqueue(sqqueue *sq) {
  sq->front=sq->rear=0;      //不能为-1
}
```

注意，循环队列头、尾指针的初值不能设为-1，但可设为 0～maxsize-1 之间的任何值，因为队列空间是循环利用的，可从其中任一位置开始使用。

2．判队空

```
int empty_sqqueue(sqqueue *sq) {
  if(sq->rear==sq->front) return 1;
  else return 0;
}
```

3．取队头

```
int gethead_sqqueue(sqqueue *sq,datatype *x) {      //队头元素值由参数返回
  if(sq->rear==sq->front) {cout<<"队空，无队头可取！\n";return 0;}
  *x=sq->data[(sq->front+1)%maxsize];  //头指针的下一个位置才是队头
  return 1;
}
```

4．入队

```
int en_sqqueue(sqqueue *sq,datatype x) {
  if((sq->rear+1)%maxsize==sq->front)
    {cout<<"队满，不能入队！\n";return 0;}//队满上溢
  sq->rear=(sq->rear+1)%maxsize;
  sq->data[sq->rear]=x;
  return 1;
}
```

5．出队

```
int de_sqqueue(sqqueue *sq,datatype *x) {           //队头元素值由参数返回
  if(sq->rear==sq->front) {cout<<"队空，不能出队！\n";return 0;} //队空下溢
  sq->front=(sq->front+1)%maxsize;
  *x=sq->data[sq->front];
  return 1;
}
```

3.2.3 队列的链接实现

队列的链式存储结构称为**链队列**，它是限制仅在表头删除和表尾插入的单链表。显然，采用尾插法的单链表比较合适。单链表由头指针唯一确定，本来不需要尾指针，但链队列经常要在尾部进行插入操作，增加尾指针后运算比较方便（但若采用尾指针表示的循环链表来表示队列，则可不需要队头指针）。

与链栈一样，队列中插入和删除操作不存在元素移动问题，采用链式存储结构，主要是避免顺序存储中存储区的预申请，或者说为了动态利用存储空间。

和单链表一样，为了运算方便，可采用头结点，这时头指针指向头结点。链队列的示意图见图 3.10。当链队列为空时，头指针和尾指针均指向头结点。

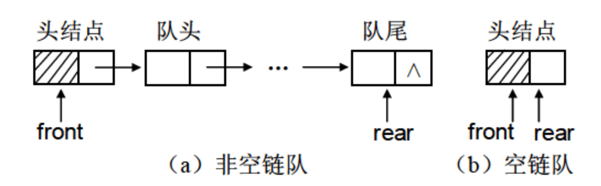


图 3.10 链队列示意图

链队列的类型定义如下：

```
typedef struct node * pointer; //结点指针类型
struct node {              //链队列结点结构
  datatype data;
  pointer next;
};
typedef struct {
  pointer front,rear;
} lkqueue;                 //链队列类型
```

下面给出链队列的 5 种基本运算。

1. 初始化

```
void init_lkqueue(lkqueue *lq) {
  pointer p;
  p=new node;              //申请头结点空间
  p->next=NULL;            //头结点 next 指针为空
  lq->front=lq->rear=p; //头指针、尾指针都指向头结点
}
```

2. 判队空

```
int empty_lkqueue(lkqueue *lq) {
  if(lq->rear==lq->front) return 1;
  else return 0;
}
```

3. 取队头

```
int gethead_lkqueue(lkqueue *lq,datatype *x) { //队头元素值由参数返回
  pointer p;
```

```
  if(lq->rear==lq->front) {cout<<"队空，无队头可取！\n";return 0;}
  p=lq->front->next;    //队头
  *x=p->data;           //取出队头元素值
  return 1;
}
```

4. 入队

```
void en_lkqueue(lkqueue *lq,datatype x) {
  pointer p;
  p=new node;           //申请新结点空间
  p->data=x;            //给新结点赋值
  lq->rear->next=p;     //原尾指针指向新结点
  lq->rear=p;           //新结点成为新尾结点
  p->next=NULL;         //新尾结点 next 指针为空
}
```

5. 出队

若当前链队列的长度大于 1，则出队时只须修改头指针的 next 域，尾指针不变，见图 3.11；但当链队列的长度为 1 时，出队后将成为空队列，这时除修改头指针外，还要修改尾指针，使它指向头结点，见图 3.12。

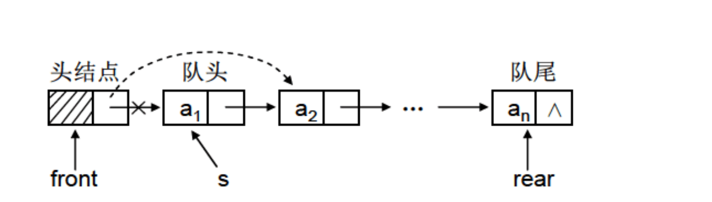


图 3.11 队列长度大于 1 时出队运算示意图

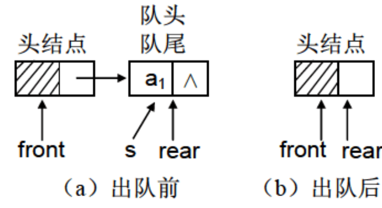


图 3.12 队列长度为 1 时出队运算示意图

出队算法如下：

```
int de_lkqueue(lkqueue *lq,datatype *x) {   //队头元素值由参数返回
  pointer s;
  if(lq->rear==lq->front) {cout<<"队空，不能出队！\n";return 0;} //队空下溢
  s=lq->front->next;                        //s 指向原队头
  *x=s->data;
  lq->front->next=s->next;
  if(s->next==NULL) lq->rear=lq->front;//仅剩一个结点时出队后要改尾指针
  delete s;
  return 1;
}
```

算法中每次出队时都要判断是否为最后一个结点出队，而一般情况下队列都不是仅剩最后一个结点，所以大多时候这种判断是多余的，效率不高。

为此可采用一种改进的等效出队算法。即出队时，删除头结点（注意，不是队头结点），使链队列的原队头结点成为新的头结点，队列的原第 2 个结点成为新的队头结点。这样，物理上删除的是头结点，逻辑上删除的是队头结点。于是，不论当前队列长度是否为 1，出队时也只修改头指针，而不用修改尾指针。指针变化情况见图 3.13。

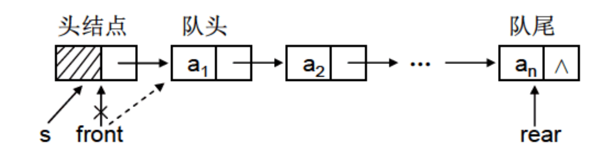


图 3.13　等效出队运算示意图

改进后的出队算法如下：

```
int de_lkqueue2(lkqueue *lq,datatype *x) {      //队头元素值由参数返回
  pointer s;
  if(lq->rear==lq->front) {cout<<"队空，不能出队！\n";return 0;} //队空下溢
  s=lq->front;           //s 指向头结点
  *x=s->next->data;      //取出原队头数据
  lq->front=s->next;     //头指针指向原队头
  delete s;              //释放原头结点
  return 1;
}
```

一般而言，具有 FIFO 特性的问题均可利用队列作为数据结构。例如，在网络中如果有多个计算机都需要通过相同的网络打印机输出结果，那么就可以按请求输出的先后次序，将这些作业排成一个队列，这就是通常所说的打印队列，凡是申请输出的作业都从队尾进入队列，每次打印的是队头，打印完后出队。

在本书的树、图等章节中，有些问题如树的层序遍历、图的广度优先遍历等就是用队列来实现的，所以这里我们就不单独列举队列应用的例子了。

最后需要指出的是，还有 2 种队列：**优先队列**（Priority Queue），其规则不是先进先出，而是优先级最高者先出；**双端队列**（Double-ended Queue），两端都可进出（但中间位置仍不能插入和删除），它可统一队列和栈，并可有更多变化，即限制两端的这 4 种操作的若干个，可分别得到队列、单栈、栈底相接的双栈、输出受限的双端队列、输入受限的双端队列等。虽然比较灵活，但实际应用并不多。



3.3　串

3.3.1　串的基本概念

串（String）是零个或多个字符组成的有限序列。一般记为 S= "$a_1a_2\cdots a_n$"（n≥0），其中 S 是串名，双引号括起来的字符序列是串值；a_i（1≤i≤n）可以是字母、数字或其他字符；串中所包含的字符个数称为该串的**长度**。长度为零的串称为**空串**，它不包含任何字符。可见，串是元素类型受限的线性表。

将串值括起来的双引号是串的定界符，它不属于串的内容，其作用是避免串与常数或标识符混淆。例如，"123"是数字字符串，它不同于整常数 123，又如"x1"是长度为 2 的字符串，而 x1 通常表示一个标识符。

串中字符的取值范围取决于所用的字符集，如 ASCII 字符集、GBK2K 字符集、Unicode 字符集等，随编码方案和字符的不同，每个字符所占用的空间可能是 1 字节到 4 字节。

在很多应用中，空格字符通常是字符集合中的一个元素，因此它可以出现在字符串中，打印出来是一个空白，有时为了清楚起见，用␣来表示它。由一个或多个空格组成的串称为**空格串**。因此“␣”和“”是不同的串，前者是长度为 1 的非空串，它含有一个空格字符，后者是长度为零的空串。

串中任意个连续的字符组成的子序列称为该串的**子串**，该串相应地称为**主串**。串中某字符在串中出现时的序号（从 1 开始）称为该字符在串中的**位置**；子串在主串中第一次出现时，子串的第一个字符在主串中的序号，称为该子串在主串中的位置。特别地，空串是任何串的子串，任何串是其自身的子串。易知，若串长度为 n，则其子串个数为 n(n+1)/2+1。

例如，有两个串 A 和 B：

A=“How␣do␣you␣do.”

B=“do”

显然 B 是 A 的子串。B 在 A 中出现了两次，但 B 在 A 中的序号是 5（首次出现的位置）。

如果两个串的长度相同，并且各个对应位置上的字符也相同，则称这两个串**相等**。

通常在程序中使用的串可分为两种：**串变量**和**串常量**。串常量和整型常数、实型常数一样，在程序中只能被引用而不能改变它们的值，一般用直接量来表示。串变量和其他类型的变量一样，其值是可以改变的，它必须用名字来识别。

早期的程序设计语言只有串常量的概念，串仅在输入或输出中以直接量的形式出现，并不参与运算。随着计算机的发展，串在文字编辑、符号处理及定理证明等许多领域得到越来越广泛的应用，这时需要将串作为一种变量，并参与一系列的运算。于是越来越多的高级语言引入了串变量的概念，并建立了一组串运算的基本函数，具有较强的串处理功能，C/C++语言更是如此。但要注意，C/C++语言并没有串变量类型，串变量是用字符数组或字符指针来间接表示的，如：

```
char name[]="John",*p="Tom";
```

其中，字符数组的内容可以改变（从而起到“变量”的作用），但数组名本身是常指针，不是变量，不能像其他变量一样对它进行赋值，比如，数组初始化后如想通过赋值语句“`name="Smith";`”来修改 name 的值是错误的（除非对“=”进行重载，如 C++等）。

3.3.2 串的基本运算

串是一种特殊的线性表，这种“特殊”不仅在于元素类型为字符，还在于其运算一般不是以“单个元素”即字符为操作对象，而是以“整体”即串为操作对象，如在串中查找某个子串、在串的某个位置上插入或删除一个子串等。具体来说，串的基本运算有 9 种。为叙述方便，本节我们假设用大写字母 S、T 等表示串（常量或变量）。

（1）赋值 ASSIGN(S, T)：加工型运算，其作用是将串 T 的值传给串 S。C/C++语言对应的是串拷贝函数 strcpy，如 strcpy(S, T)。

（2）联接 CONCAT(S, T)或 CONCAT(S, T, V)：其作用是由 S 和 T 连接成一个新串，其中 T 对应的串值紧接着放在 S 的后面。前者为加工型运算，新串存放在原来 S 的位置上；后者为引用型运算，新串存放在 V 中。C/C++语言对应的是串联接函数 strcat，如 strcat(S, T)。

例如：

```
CONCAT("Th","is")="This"
```

（3）求串长 LENGTH(S)：引用型运算，运算结果是串 S 的长度。C/C++语言对应的是串长函数 strlen，如 strlen(S)。

（4）求子串 SUBSTR(S, i, j)：引用型运算，运算结果是串 S 中第 i 个字符开始连续 j 个字符组成的子串。其中参数应满足：1≤i≤LENGTH(S)，0≤j。例如：

```
SUBSTR("who", 2, 2)="ho",
SUBSTR("this", 3, 0)="",
SUBSTR("there", 4, 5)="re"
```

其中，最后一例从原串中第 4 个位置开始没有 5 个字符可取，则取到串尾结束，这时该子串共有 LENGTH−i+1 个字符。

（5）串比较 COMPARE(S, T)或判等 EQUAL(S, T)：引用型运算，前者是比较两个串 S 和 T 的大小，运算结果小于、等于或大于 0，分别表示 S<T、S=T 和 S>T。后者比较两个串 S 和 T 是否相等，结果为 1（相等）或 0（不相等）。C/C++语言对应的是串比较函数 strcmp，如 strcmp(S, T)。

串的大小通常是按字典序定义的，即从两个串的第 1 个字符起，逐个比较相应的字符，直到找到两个不等的字符为止，由这两个不等的字符来确定串的大小。例如，“this”>“there”，这是因为“i”>“e”。若找不到两个不等的字符，就必须由串长来决定大小。例如“there”>“the”，这是因为两个串的前三个字符均一一相同，但前者长度大于后者。

这里字符的大小由该字符在字符集中出现的先后次序确定。在常用字符集中，数字字符 0～9、字母字符 A～Z（或者 a～z）等各自是顺序排列的。汉字的大小按编码确定。汉字的编码有几种，如我国的国标码（GB2312）、中国台湾地区的 Big5 等。对 GB2312，一级字库（常用汉字）中汉字的国标码之间的大小关系，与对应的汉语拼音构成的串的大小关系一致，而二级字库中汉字编码之间的大小关系，与对应汉字的笔划数目的大小关系一致。另外，小写字母的编码大于大写字母，汉字的编码大于字母等，因此有：

空格<…<“0”～“9”<…<“A”～“Z”<…<“a”～“z”<…<汉字

（6）插入 INSERT(S, i, T)：加工型运算，其作用是将串 T 插入到串 S 的第 i 个字符位置。其中参数应满足：1≤i≤LENGTH(S)+1。例如：

```
INSERT("is", 1,"Th")="This",
INSERT("da", 3,"ta")="data"
```

（7）删除 DELETE(S, i, j)：加工型运算，其作用是从 S 中删除第 i 字符开始的连续 j 个字符。其中参数应满足：1≤i≤LENGTH(S)，0≤j。例如：

```
DELETE("Good", 3, 2)="Go",
DELETE("Good", 3, 0)="Good"
```

（8）子串定位 INDEX(S, T)：引用型运算，其作用是在主串 S 中查找是否有等于 T 的子串，若有则返回 T 在 S 中第一次出现的位置或指针；否则返回零。显然 T 不能为空串。C/C++语言对应的是子串定位函数 strstr，如 strstr(S, T)。还有一个类似的字符定位函数 strchr，如 strchr(S, 'a')。例如：

```
INDEX("Good","od")=3,
```

```
INDEX("Good","do")=0
```

（9）置换 REPLACE(S, i, j, T)或 REPLACE(S, T, R)：加工型运算，前者作用是用 T 置换 S 中第 i 个字符开始的连续 j 个字符；后者作用是用 R 替换所有在 S 中出现的、和 T 相等的子串。例如：

```
REPLACE("Who", 3, 1,"ere")="Where"
REPLACE("How␣do␣you␣do.","do","DO")="How␣DO␣you␣DO."
```

严格说来，上述运算中只有前 5 个是基本运算，后面几个不是。比如利用置换运算 REPLACE(S, i, j, T)可完成插入和删除功能：REPLACE(S, i, 0, T)即在串 S 中第 i 个位置插入串 T；REPLACE(S, i, j, “”)即在串 S 中删除从位置 i 开始的 j 个字符。

又如，我们可利用判等、求串长和求子串等运算实现子串定位 INDEX(S, T)：依次从主串 S 中第 i（i≥1）个字符开始，取出长度与 T 相同的子串和 T 比较，若相等则子串位置为 i；否则 i 增 1 取下一个子串比较，直到最后找到或没有。算法如下：

```
int INDEX(string S,string T) { //假设 string 为串的类型
  int m,n,i;
  m=LENGTH(T);if(m==0) return 0;//T 为空串
  n=LENGTH(S);
  for(i=1;i<=n-m+1;i++)          //主串搜索的合法位置为 1～n-m+1，其后至少 m 个字符
    if(EQUAL(SUBSTR(S,i,m),T)) return i;
  return 0;                      //S 中没有子串 T
}
```

顺便指出，子串定位又称串的**模式匹配**（Pattern Matching）或**串匹配**（String Matching），其中子串称为**模式串**。上述算法称为朴素（或简单）模式匹配算法，虽然简单，但效率不高，最坏时间复杂度为 O((n−m+1)m)，若 n≫m，则为 O(mn)。效率高的算法也有，但一般较复杂，如著名的 KMP（Knuth-Morris-Pratt）算法，它注意到匹配不成功时，子串中已比较成功的字符也就是主串的相应字符，分析子串的组成可知主串可从当前位置接着再和子串的某个位置继续比较（主串位置不回退），效率可提高到 O(m+n)，具体见 3.3.4 节。

之所以将上述串运算定义为基本运算进行单独实现，而不通过其他运算，是因为它们就像数值计算中对整型变量进行四则运算那样，频繁地用在串处理中。因此，在引进串变量（或等效串变量）的高级语言中，一般都将它们作为基本运算符或基本内部函数来提供，当然，提供的种类和符号在各个语言中可能有所不同。C/C++语言提供了丰富的串函数，其函数原型可参见头文件 string.h，上面我们仅提到了其中比较常用的几个。

3.3.3 串的存储结构

存储串的方法也就是存储线性表的一般方法，不过由于组成串的结点是单个字符，所以在具体存储时有一些特殊的技巧，下面分别介绍。

1．顺序存储

串的顺序存储结构简称为**顺序串**，即将串中的字符顺序地存放在内存中一片连续的存储单元中。

一般来说，一个字节（8 位二进制数）就可以表示一个字符（即该字符的 ASCII 码）。如果存储单元是按字编址的，则一个内存单元有多个字节，可以存放多个字符，如 32 位的

内存单元可以存储 4 个字符。这时，如果一个内存单元仍只存放一个字符，这种存储方式就称为非紧缩格式，见图 3.14，其中斜线部分表示空闲字节；如果一个内存单元存放多个字符，这种存储方式就称为紧缩格式，见图 3.15。显然，紧缩格式可以节省存储单元，但对串值进行访问时，需花费额外的时间分离同一存储单元中的字符。

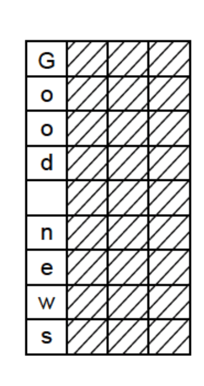


图 3.14 非紧缩格式示意图

G	o	o	d
	n	e	w
s			

图 3.15 紧缩格式示意图

由于 C/C++语言可以按字节寻址，于是每个字符在内存中占一个字节，串中相邻的字符便顺序存放在相邻的字节单元中，这样既节约空间，处理也很方便。

串是一个字符序列，为了表示串的结束，可以用一个特定的、不会在串中出现的字符作为串的终结符，放在串值的尾部。在 C/C++语言中用字符“\0”作串的终结符，如串“good␣news”的顺序存储结构如图 3.16 所示。这时顺序串可用字符数组来描述：

```
const int maxsize=100;  //假设串可能的最大长度是 100
char ch[maxsize+1];     //数组的大小要考虑额外加上的终结符
```

0	1	2	3	4	5	6	7	8	9	…	maxsize−1
g	o	o	d		n	e	w	s	\0	…	

图 3.16 C/C++语言中顺序串存储结构示意图

若不设置终结符，还可用一个整数 n 来指示串的长度，这时顺序串的类型定义和顺序表类似：

```
const int maxsize=100;  //假设串可能的最大长度是 100
typedef struct {
  char ch[maxsize];     //串的存储空间，这时串不需要终结符
  int n;                //当前串的长度
} sqstring;
```

和顺序表类似，顺序串上的插入、删除操作不方便，操作中可能需要移动大量的字符。

2. 链式存储

串的链式存储结构简称为**链串**。链串的组织形式与一般的单链表类似，但链串的一个存储结点可存储多个字符。通常将链串中每个存储结点所存储的字符个数称为结点的大小。链串的类型定义如下：

```
const int nodesize=4;           //结点大小，假设为 4
typedef struct node * pointer;  //结点指针类型
struct node {
  char S[nodesize];
  pointer next;
};
typedef pointer lkstring;       //链串类型
```

与顺序串类似，如果一个结点只存储一个字符（结点大小为 1），就称为非压缩形式；如果一个结点存储多个字符(结点大小大于 1)，就称为压缩形式。例如，对串 S=“do␣work”，图 3.17（a）、（b）分别表示了结点大小为 1 和 4 的两个链串。

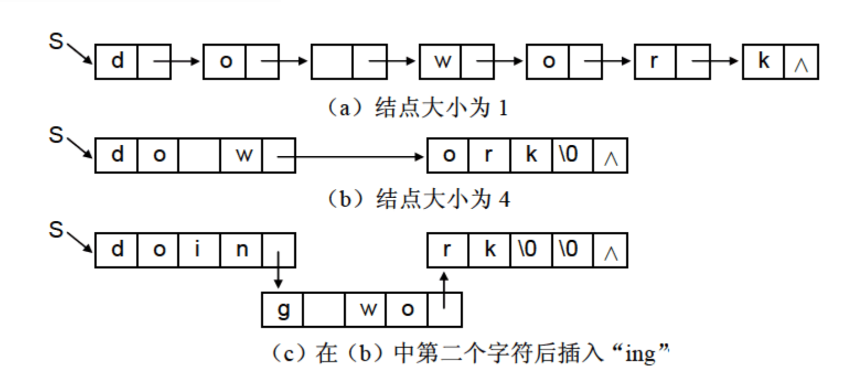


图 3.17 链串的存储结构示意图

结点大小为 1 的链串空间利用率较低，因为结点的存储密度低，如设每个字符占 1 个字节，指针占 4 个字节，则存储密度只有 20%。提高结点的大小可提高存储密度，如结点大小为 4 时，存储密度便达到 50%。但结点大小大于 1 后又会出现新的问题，如串的长度不一定正好是结点大小的整数倍，需用特殊字符（例如 "\0"）来填充最后一个结点，以表示串的终结；在插入、删除运算时，可能会引起大量字符的移动，给运算带来不便。图 3.17（c）表示在串 S 的第 2 个字符后插入 "ing" 时，要移动前后两个结点中的 5 个字符。

显然，结点大小大于 1 的链串，可看成是一种顺序与链式相结合的结构。

3．索引存储

该方法是用串变量的名字作为关键字组织索引表（名字表），索引表中的地址部分除了要指出串存放的起始地址外，还必须有信息指出串存放的末地址。末地址的表示方法一般有几种：给出串长、给出尾指针、在串尾设置结束符等。

索引表一般是有序的并且顺序存放，串值数据一般也顺序存放。图 3.18（a）～图 3.18（c）表示了这 3 种索引方式下串的存储结构，其中的两个串是 S1= "where"，S2= "you"。

以上第三种方式还有一种变形，即当串很短，不超过一个指针空间时，就将它存放在指针域 start 中。这时为了区分 start 域到底是地址还是串值，需要在索引表中增加一个标志位 tag，称为带特征位的名字表，如图 3.18（d）所示。

在串的顺序存储表示中，串值空间的大小是在程序说明部分定义的；在程序运行期间串的长度变化范围不能超过它，否则会产生溢出（尤其在进行联接、置换等运算时）。如果一个程序要使用很多串，对每个串都分配一个可能的最大空间，显然空间浪费很大。如果采用链串进行动态存储，则结点大了运算不方便，结点小了空间利用率又不好。

较好的解决办法是，对串采用一种称为**堆结构**[①]的动态存储结构：在系统中开辟一个容量很大、地址连续的存储空间作为存放串值的可利用空间。当建立一个新串时，就从该空间中分配一个大小和串的长度相同的、地址连续的存储空间用于存储新串的值。这样所有串的串值都存储在这个可利用空间中，同时为每个串建立一个索引，以记录该串的长度以及该串值在可利用空间中的起始位置。这样，利用串的索引存储方法，可实现多个串值空间的共享和动态分配。

① 一般把动态存储区分为（组织成）栈区和堆区，前者用于具有后进先出特点的数据，后者用于无后进先出特点的数据。

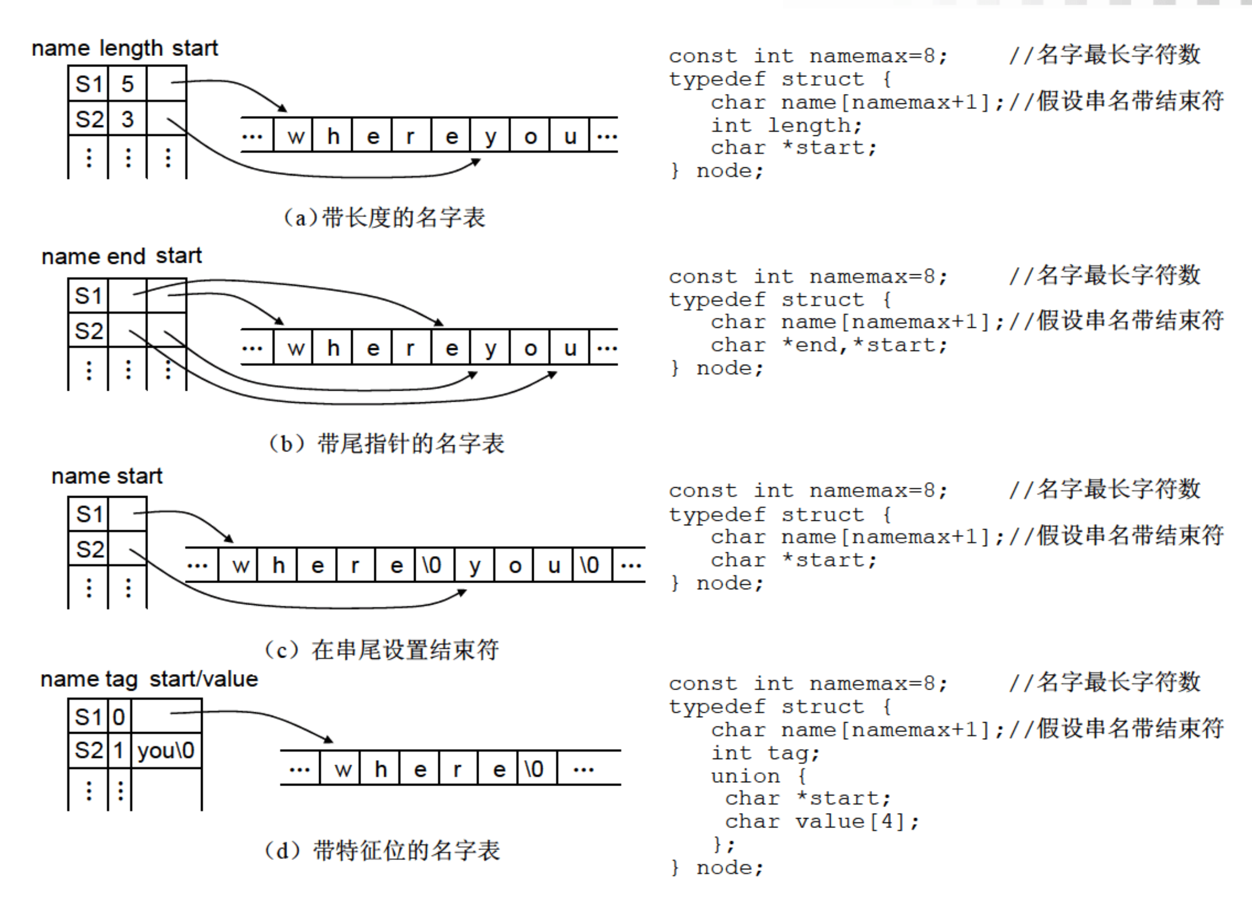


图 3.18 串的索引存储结构示意图

例如，在文本编辑程序中，可以为整个文本建立一个文本缓冲区（堆），文本的每一行可看成一个字符串，并为每一行建一个索引。此时索引表的名字域就是行号，故又称为**行表**。若干行组成一页，也可为每一页建一个索引得到**页表**。在给一个串分配空间时，是在当前堆中的空闲位置处进行的，故需要一个指针指示当前空闲区位置，它的初值为 0，每分配完一个串后，就修改该指针。在具体使用时，还有些问题需要解决，如已删除串的空间再利用、在串中插入时如何扩充它的空间等。

在 C/C++语言中，就存在一个称为堆的动态存储区，由 C 的系统函数 malloc( )和 free( )或 C++的操作符 new 和 delete 来动态管理。

例 3.5 编写算法，比较两个串的大小，不要使用系统函数 strcpy。

解：两个串的比较就是从头开始逐个字符比较，若全部相等则返回 0，否则返回当前不等字符的 ASCII 码之差（若某个串先比较完，则可视其当前字符为空）。

如果是顺序串，并用上述 sqstring 类型表示，则算法如下：

```
int comp(sqstring *s1,sqstring *s2) {
  int i,m;
  if(s1->n<s2->n) m=s1->n;             //找较短的长度
  else            m=s2->n;
  i=0;
  while(i<m && s1->ch[i]==s2->ch[i]) i++;
  if(i==s1->n && i==s2->n) return 0;    //两串相同
  else if(i==s1->n) return -s2->ch[i]; //s1 串较短
```

```
  else if(i==s2->n) return  s1->ch[i]; //s2 串较短
  else return s1->ch[i]-s->ch2[i];
}
```

如果是链串，且结点大小为 1，则算法如下：

```
int comp(lkstring S,lkstring T) {
  pointer p,q;
  p=S;
  q=T;
  while(p!=NULL && q!=NULL && p->ch==q->ch) {p=p->next;q=q->next;}
  if(p==NULL && q==NULL) return 0;     //两串相同
  else if(p==NULL) return -q->ch;      //S 串较短
  else if(q==NULL) return  p->ch;      //T 串较短
  else return  p->ch-q->ch;
}
```

以上两种算法，运算量和难易程度相当。但是，如果采用带结束符“\0”的顺序串，则算法显得非常简便：

```
int comp(char s1[],char s2[]) {
  int i;
  i=0;
  while(s1[i]!='\0' && s1[i]==s2[i]) i++;
  return s1[i]-s2[i];
}
```

串的一些其他运算，如联接、拷贝等，用带结束符的顺序串实现时，也都比较简便。所以，串在实际使用时，经常采用带结束符的顺序串。

本节讨论了串的几种最基本的存储表示方法，对于具体的应用问题，还可以根据实际情况设计出更为合理有效的（组合型）存储方法。

3.3.4* 串的模式匹配

串的模式匹配（或称子串定位）是各种串处理系统中最重要的运算之一，对其效率的改进和提高有重要的实际意义。人们提出了许多效率不同的算法，以下介绍两种算法，假设串的储存结构为上节定义的顺序串。

1．BF 算法

在 3.3.2 节给出了用串的其他运算实现的简单模式匹配算法，基本思想是依次从主串的各字符位开始与子串进行匹配，直到匹配成功或最终失败。为了提高效率，可写出不依赖其他串运算的匹配算法，下面就是其中一种实现：

```
int index(sqstring *s,sqstring *t) {//BF 算法
  int i,j;
  i=0,j=0;
  while(i<s->n && j<t->n) {
    if(s->data[i]==t->data[j]) {i++;j++;}//对应字符相等时向后推进
    else  {i=i-j+1;j=0;}              //i 退到主串下一趟开始位置，j 重新开始
  }
  if(j>=t->n) return i-t->n;
  else return -1;
}
```

该算法常称 Brute-Force 算法（简称 BF 算法），其中每次匹配失败时，主串指针 i 要退到下一趟的开始位置。算法的最好时间复杂性为 O(m)，最坏时间复杂性为 O(nm)，其中 n、m 分别为主串、模式串的长度。

2. KMP 算法

这是一种改进的串匹配算法，由 D. E. Knuth 与 V. R. Pratt 和 J. H. Morris 同时发现[①]，常称 KMP 算法。该算法的时间复杂度为 O(n+m)。其基本思想是：每当匹配失败时，主串指针 i 不回退，而是根据已匹配的信息将模式向右“滑动”一定位置后继续进行比较。

设比较失败时主串、模式串位置分别为 i、j，即 $s_i \neq t_j$，之前“$t_0t_1 \cdots t_{j-1}$”=“$s_{i'}s_{i'+1} \cdots s_{i-1}$”，其中 i'（=i−j）为本趟比较开始时的主串位置，见图 3.19（a）所示。若下趟从主串的下一位置 i'+1（=i−j+1）开始比较，则首先比较 $t_0=s_{i'+1}$ 是否成立，由于 $s_{i'+1}=t_1$，这相当于比较 t_0 和 t_1，若 $t_0 \neq t_1$ 则本趟比较肯定失败；类似，若 $t_0 \neq t_2$，则从主串位置 i'+2（=i−j+2）开始比较也肯定失败……。

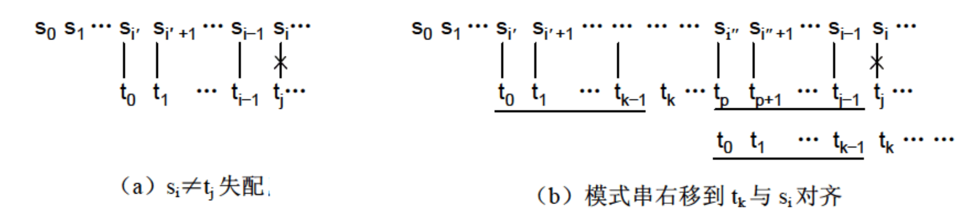


图 3.19　KMP 算法原理

一般地，若模式串某位置 p（p≤j−1）满足 $t_0=t_p$，则主串至少可从与 t_p 对应的位置 i"（=i'+p）开始比较（若这种位置有多个，则先考查前面的）。但这里只考虑了比较的第一个字符。显然，若比较能进行下去，还应有 $t_1=t_{p+1}$，$t_2=t_{p+2}$，…。易见，这种相等关系应一直延续到 t_{j-1}，否则该趟比较仍失败，需找下一个与 t_0 相等的位置，再进行同样的处理……。

设前后共有 k 个字符对应相等，即“$t_0t_1 \cdots t_{k-1}$”=“$t_pt_{p+1} \cdots t_{j-1}$”，如图 3.19（b）所示，本趟比较主串可从位置 i"（=i'+p）开始，但“$s_{i''} \cdots s_{i-1}$”都与模式串中“$t_0 \cdots t_{k-1}$”对应字符相等，这部分比较并不需要真的进行，只需从 s_i 开始与模式串的 t_k 开始比较即可。可见，比较失败时主串位置 i 不需回退，记 next[j]=k，则下次继续和 $t_{next[j]}$ 比较。这相当于比较失败时将模式串右移到 t_k 和 s_i 对齐后继续比较。

“$t_0t_1 \cdots t_{k-1}$”=“$t_pt_{p+1} \cdots t_{j-1}$”的含义是在失配字符 t_j 之前，模式串的首尾 k 个字符对应相等。若这种子串有多个，则取最长的一个（尾部串从可能的最前面的位置开始）。

若 t_j 之前没有首尾相同的子串，则取 k=0，即模式串从 t_0 开始与主串 s_i 开始进行比较。若这时 $t_0 \neq s_i$，则进行新一趟的比较，即从 t_0 与 s_{i+1} 开始比较。

如上所述，next[j]只取决于模式串，反映模式串本身的局部匹配信息，一般定义如下[②]：

$$\text{next}[j]=\begin{cases} -1 & \text{当} j=0 \text{时} \\ \max\{k \mid 0<k<j \text{ 且 “} t_0 \cdots t_{k-1} \text{”=“} t_{j-k} \cdots t_{j-1} \text{”}\} & \text{当此集合非空时} \\ 0 & \text{其他情况} \end{cases}$$

① 1970 年 Cook 的一个理论结果表明有算法能在大约 m+n 时间内解决模式匹配问题。D. E. Knuth 和 V. R. Pratt 在重建 Cook 的证明时创建了这个模式匹配算法；大概同一时间 J. H. Morris 在设计文本编辑器时也创建了差不多同样的算法。

② 也可令 j=0 时 next[0]=0，这时需对 KMP 算法（以及求 next 的算法）语句略作调整，具体略。

KMP 算法如下：

```
int KMP(sqstring *s,sqstring *t) {//KMP 算法
  int i,j;
  int next[MMAX];
  NEXT(t,next);                    //求模式串 t 的 next 数组
  i=0,j=0;
  while(i<s->n && j<t->n) {
    if(j==-1 || s->data[i]==t->data[j]) {i++;j++;}
    else  j=next[j];
  }
  if(j>=t->n) return i-t->n;
  else return -1;
}
```

算法的主要运算是字符比较，从 while 循环可见，该运算量不超过 if 条件测试次数，而后者等于 i、j 前进（i++;j++）次数（条件成立时）与 j 回退（j=next[j]）次数（条件不成立时）之和。由于 i<n，故 i 前进次数为 O(n)。j 随 i 同步前进，故 j 前进次数也为 O(n)。注意 j 前进时每次为 1 位、回退时每次可多位，而前进、回退的总结果，即最终匹配成功或失败时 j≥0，所以 j 回退次数不超过前进的次数 O(n)。于是字符比较（以及整个 while 循环）的时间复杂度为 O(n)。另外，求 next 数组（见下文所述）的时间复杂度为 O(m)，所以 KMP 算法的时间复杂度为 O(n+m)。

下面看看 next 数组的求法。若按定义直接对每个位置都进行前后重复子串的测试，显然效率不高，这里采用递推方法。

首先由定义得 next[0]= −1。设已求出 next[j]=k，即“$t_0t_1\cdots t_{k-1}$”＝“$t_pt_{p+1}\cdots t_{j-1}$”，若下一字符 $t_k=t_j$，则“$t_0t_1\cdots t_k$”＝“$t_pt_{p+1}\cdots t_j$”，即下一位置的前后重复串长比当前位置的多 1，所以 next[j+1]=k+1= next[j]+1。

若 $t_k\neq t_j$，这时若把 s'=“$t_0t_1\cdots t_pt_{p+1}\cdots t_j$”看成主串，t'=“$t_0t_1\cdots t_k$”看成模式串，问题变为两者在 $t_j\neq t_k$ 之前已匹配“$t_pt_{p+1}\cdots t_{j-1}$”＝“$t_0t_1\cdots t_{k-1}$”，与前述 KMP 算法推导类似，这时主串位置 j 不动，继续和模式串的 k'=next[k]处比较（模式串右移）。于是，重复前面的过程，若 $t_{k'}=t_j$ 则 next[j+1]=k'+1=next[k]+1；若 $t_{k'}\neq t_j$ 则主串位置 j 不动，继续和模式串的 k"=next[k']处比较（模式串右移）……，依次类推，直到 t_j 和模式串的某字符匹配成功，或者始终不匹配则令 next[j+1]=0。

可见，求 next 数组的过程相当于模式串的“自我匹配”，算法与前述 KMP 算法类似：

```
void NEXT(sqstring *t,int next[]) {//求 next
  int j,k;
  j=0;k=-1;next[0]=-1;
  while(j<t->n-1) {
    if(k==-1 || t->data[j]==t->data[k]) {
      j++;k++;
      next[j]=k;
    }
    else k=next[k];
  }
}
```

类似前述 KMP 算法的分析，该算法的时间复杂度为 O(m)。

这里的 next 数组尚可改进：在模式匹配中，设 next[j]=k，若 $t_j=t_k$，则在 $s_i\neq t_j$ 时必有

$s_i \neq t_k$，即后一比较是多余的，可直接和 $t_{next[k]}$ 比较，即此时可令 next[j]=next[k]。改进算法如下（匹配算法不变）：

```
void NEXTVAL(sqstring *t,int nextval[]) {//求改进的 next
  int j,k;
  j=0;k=-1;nextval[0]=-1;
  while(j<t->n-1) {
    if(k==-1 || t->data[j]==t->data[k]) {
      j++;k++;
      if(t->data[j]==t->data[k]) nextval[j]=nextval[k];
      else nextval[j]=k;
    }
    else k=nextval[k];
  }
}
```

例 3.6　设目标串 s=“abcaabbabcabaacba”，模式串 t=“abcabaa”，画出 BF 算法、KMP 算法、改进 KMP 算法的模式匹配过程示意图，它们分别要进行多少趟匹配？

解：对 KMP 算法要先求出 next、nextval 函数值，见表 3.1 所示。匹配过程如图 3.20 所示，三种算法所需的匹配趟数分别为 8、5、4 趟，其中 KMP 算法在 next 数组改进前多了一趟无用的比较（虽然只比较了一个字符）。

表 3.1　模式串的 next、nextval 函数值

j	0	1	2	3	4	5	6
模式 t	a	b	c	a	b	a	a
next[j]	–1	0	0	0	1	2	1
nextval[j]	–1	0	0	–1	0	2	1

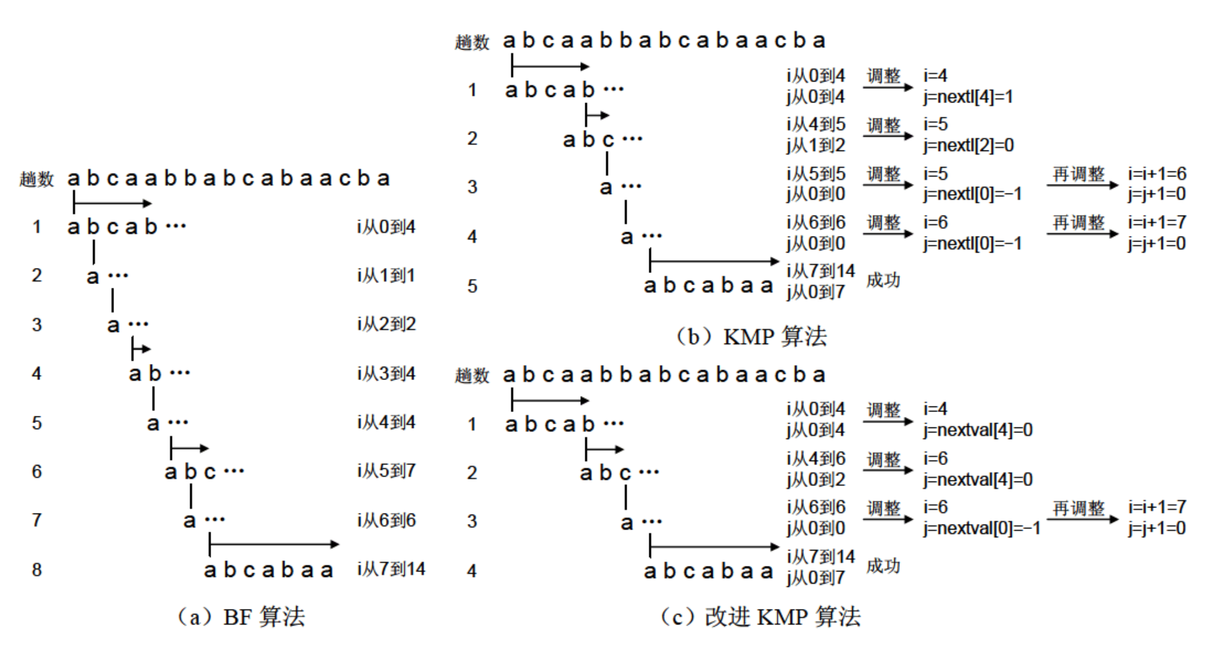


图 3.20　三种算法下的模式匹配过程

最后需要指出，以上在 KMP 算法的叙述中，串元素序号是从 0 开始的，这与 C/C++ 数组下标从 0 开始一致（应用到 C/C++语言的以“\0”结尾的顺序串也非常方便）；若从 1

开始，则各字符对应的 next 值要比这里多 1[①]，而串字符的存储，比如 s_i，要么存放到 s[i−1]，要么仍存放到 s[i]（0 号单元不用或作它用）；有关算法语句也需略作调整，具体略（参见习题 3.21）。

习　题　三

3.1　设{A, B, C, D, E, F}依次进栈，出栈序列为{B, D, C, F, E, A}，则栈的容量至少为多少？

3.2　证明：

（1）设入栈序列为{1, 2, …, n}，则出栈序列不可能为{…, k, …, i, …, j, …}，其中 i<j<k。

（2）*设入栈序列为{1, 2, …, n}，则所有可能的出栈序列个数为 $\frac{1}{n+1}C_{2n}^{n}$ 。

3.3　能否用栈实现队列的功能？

3.4　设两个顺序栈共享空间，试写出两个栈公用的栈操作算法 push(x, k)和 pop(k)，其中 k 为 0 或 1，用以指示栈号。

3.5　设一批数据有正有负，试用栈对它们进行调整，使输出时所有负数都在正数之前。

3.6　设计算法，判断一个算术表达式中的圆括号是否正确配对。

3.7　设计算法，借助栈将单链表逆置。

3.8　设单链表中存放着 n 个字符，编写算法，判断该字符串是否有中心对称关系（又称回文），例如 xyzzyx、xyzyx 都是中心对称的字符串。

3.9*　试设计递归算法，计算 1+2+3+…+n。

3.10　设用尾指针表示的带头结点的循环链表来表示队列，试写出出队和入队算法。

3.11　设循环队列为 A[0..m−1]，队头指针 front 指向队头的前一个位置，队尾指针 rear 指向队尾，队列元素个数为 len。若已知 front、rear 和 len 中的两个，则另一个为多少？

3.12　设循环队列为 A[0..m−1]，队头指针 front 指向队头的前一个位置，队尾指针 rear 指向队尾。分别对下列各种情况，写出出队和入队算法：

（1）不设队头指针 front，而设队尾指针 rear 和队列元素个数 len。

（2）不设队尾指针 rear，而设队头指针 front 和队列元素个数 len。

（3）除了队尾指针 rear 和队头指针 front 外，另设一个表示队空或满的标志位 flag。

3.13　设循环队列为 A[1..m]，队头指针 front 指向队头的前一个位置，队尾指针 rear 指向队尾。写出队列运算的主要语句。

3.14　设循环队列为 A[0..m−1]，若将队头指针 front 指向队头（而不是其前一个位置），队尾指针 rear 指向队尾。写出队列运算的主要语句。

3.15　设某栈中元素依次为{A_1, A_2, …, A_n}，A_n 为栈顶；某队列中元素依次为{B_1, B_2, …, B_n}，B_1 为队头。现要将栈中元素调整到队列中，使队列中元素依次为{B_1, A_1, B_2, A_2,

① 这只是表面上不同，实际执行过程是相同的，比如从 0 开始编号时某字符 x 的 next 为 k，即下次要退到编号为 k 的字符 y，而从 1 开始编号时该字符 y 的编号正是 k+1。

B_3, A_3, …, B_n, A_n}。试给出一种实现方法，并分析其运算量（每次出、入栈或队列算 1 次基本运算）。

3.16　简述下列每对术语的区别：空串和空格串；串变量和串常量；主串和子串；串名和串值。

3.17　若两个串 S 和 T 的连接 concat(S, T)=concat(T, S)，试分析所有可能的 S 和 T。

3.18　编写算法，将两个串连接起来，不要使用系统函数 strcat。

3.19　编写算法，将串 S2 拷贝到串 S1 中，不要使用系统函数 strcpy。

3.20　设 S、T 是两个结点大小为 1 的链串，编写算法，找出 S 中第一个不在 T 中出现的字符。

3.21*　对如下储存结构，试写出 KMP 算法：

```
#define MMAX 100            //MMAX 为串长上限(不能超过 1 字节最大整数 255)
typedef char sstring[MMAX+1];//0 号单元存放串长(不能超过 MMAX)
```

3.22*　设模式串为 t= “abcaacabaca”，写出其 next、nextval 数组值。

第4章 多维数组和广义表

前面几章介绍的线性表、栈、队列和串都是线性结构，本章将要介绍多维数组和广义表，它们是复杂的非线性结构，因为所选内容不多，放在同一章中讨论。

多维数组中的元素同时属于多个线性表，可认为是一种广义的线性表。数组几乎是和程序设计语言同时诞生的，但在程序设计语言中，重点是数组的使用，本章则是介绍数组的内部实现，主要是数组的存储方式与寻址，以及矩阵的压缩存储。

广义表是一种特殊的数据结构，它兼有线性表、树、图等结构的特点。从各层元素各自具有的线性特征方面看，它应该是线性表的推广；从元素的分层方面看，它有树结构的特点；但从元素的递归性和共享性等方面看，它应该属于图结构。总之，它是一种更为复杂的非线性结构。本章只介绍广义表的基本概念和储存结构。

4.1 多维数组

数组（Array）是一种十分常用的结构类型，程序设计语言一般都直接支持数组类型。数组中各元素的类型相同，并且元素的个数和元素间的关系在数组建立后一般不能改变。

多维数组可看成线性表的推广。对一维数组，它就是一个线性表；对二维数组，可看成每个元素为一维数组的线性表，这个元素可以是行向量，也可以是列向量，示意见图 4.1。

$$A_{m\times n}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

（a）二维数组

$$A_{m\times n}=\left[\begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix}\begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix}\cdots\begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix}\right]$$

（b）n 个列向量

$$A_{m\times n}=\begin{bmatrix} [a_{11} & a_{12} & \cdots & a_{1n}] \\ [a_{21} & a_{22} & \cdots & a_{2n}] \\ & \vdots & & \\ [a_{m1} & a_{m2} & \cdots & a_{mn}] \end{bmatrix}$$

（c）m 个行向量

图 4.1　二维数组

类似地，一个三维数组可看成每个元素为二维数组的线性表。一般地，一个 n 维数组可看成每个元素为 n−1 维数组的线性表。与此类似，在 C/C++语言中，对多维数组，除了可以直接定义外，还可以这样定义：将二维数组定义为每个元素为一维数组的数组，将三维数组定义为每个元素为二维数组的数组等。例如下列定义实际上定义了二维数组

int x[m][n]:

```
const int m=10;
const int n=5;
typedef int R[n];
R x[m];              //x 有 m 个元素，每个元素是 R 类型（n 个元素的一维数组）
```

多维数组的每个元素受多个线性关系的约束，可有多个直接前趋和直接后继，故是非线性结构。例如，二维数组中每个元素既在某一行上，也在某一列上，而每一行、每一列上的元素都是线性关系，于是在整个数组中，每个元素可有两个直接前趋和两个直接后继（行、列方向各一个），从而是非线性结构。当然，首元素 a_{11} 没有前趋、尾元素 a_{mn} 没有后继；第一列/行元素没有行/列方向的前趋，最后一列/行元素没有行/列方向的后继。

数组通常只有两种基本运算——读和写：

（1）读。给定一组下标，读出相应的元素。

（2）写。给定一组下标，修改相应的元素。



4.2 数组的存储结构

数组是一种在高级语言中已经实现了的数据结构：其类型定义由高级语言的数组类型直接给出；由于没有插入和删除运算，存储结构采用的是顺序存储方式；读写通过下标定位和赋值运算来完成。下面只讨论其中下标定位的原理——寻址问题，并假设数组元素的下标是有效的（注意 C/C++语言并不检测数组下标的合法性）。

在讨论地址问题时，一般取数组开始结点的地址作为基准（基址），然后考察其他结点相对此基址的偏移量（偏移地址）。显然，基址加上任一结点的偏移量就是该结点的绝对地址。

一维数组的每个元素只含一个下标，其实质就是线性表，采用顺序存储时与线性表的顺序存储结构（顺序表）基本相同，即将数组各元素按它们的逻辑次序依次存储到一片连续的存储单元中，但这里元素的个数是固定的，不能改变。设一维数组为 $A=[a_0, a_1, \cdots, a_{n-1}]$，元素 a_0 的地址为 LOC(0)，每个元素占用的存储单元数为 c，则元素 a_i 的地址 LOC(i)为：

$$LOC(i)=LOC(0)+i\times c$$

如果数组元素的下标从 1 开始，$A=[a_1, a_2, \cdots, a_n]$，则元素 a_i 的地址为

$$LOC(i)=LOC(1)+(i-1)\times c$$

一般地，若数组元素的下标范围为[s, t]，即 $A=[a_s, a_{s+1}, \cdots, a_t]$，则元素 a_i 的地址为：

$$LOC(i)=LOC(s)+(i-s)\times c$$

二维数组的每个元素含有行、列两个下标，例如：

$$A_{m\times n}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \ddots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

是一种非线性结构，但内存单元是一维的线性结构，在进行顺序存储时，需要将多维关系映射为一维的线性关系，常用的方法有如下两种。

1．按行存放（行优先，Row-major Ordering）

在这种方法中，从数组的第一行开始，每一行按从左到右的顺序（列号递增）对数组元素依次存放。例如，对上面的二维数组 A，按行存储的元素次序为：

$a_{11}, a_{12}, \cdots, a_{1n}, a_{21}, a_{22}, \cdots, a_{2n}, \cdots, a_{m1}, a_{m2}, \cdots a_{mn}$

在 Pascal、C/C++语言中，数组就是按行存储的。

设二维数组的一般形式为 A[s..t, q..r] ①，这里 s 和 q 不一定是 1 或 0。对元素 a_{ij}，它前面一共有 i−s 行，每一行有 r−q+1 个元素（即列数），第 i 行从开始到 a_{ij} 为止有 j−q+1 个元素，故从数组开始到 a_{ij} 为止的元素个数为：

$h=(i-s)\times(r-q+1)+(j-q+1)$

相应地，元素 a_{ij} 的地址 LOC(i, j)为：

$LOC(i, j)=LOC(s, q)+(h-1)\times c=LOC(s, q)+[(i-s)\times(r-q+1)+(j-q)]\times c$

其中，c 为每个元素所占的存储单元数。

例如，对数组 A[1..m, 1..n]，元素 a_{ij} 的地址为：

$LOC(i, j)=LOC(1, 1)+[(i-1)\times n+(j-1)]\times c$

特别地，对 C/C++语言数组 A[m][n]，即 A[0..m−1, 0..n−1]，元素 a_{ij} 的地址为：

$LOC(i, j)=LOC(0, 0)+[i\times n+j]\times c$

2．按列存放（列优先，Column-major Ordering）

在这种方法中，从数组的第一列开始，每一列按从上到下的顺序（行号递增）对数组元素依次存放。例如，对上面的二维数组 A，按列存储的元素次序为：

$a_{11}, a_{21}, \cdots, a_{m1}, a_{12}, a_{22}, \cdots, a_{m2}, \cdots, a_{1n}, a_{2n}, \cdots, a_{mn}$

在 FORTRAN 语言中，数组就是按列存储的。

上述规则可推广到多维数组的情形：

（1）按行存储时，对每一个左下标，逐个变动其右边的下标（简称右下标先动），或者说，先变动最右的下标，从右到左，最后变动最左的下标。

（2）按列存储时，对每一个右下标，逐个变动其左边的下标（简称左下标先动），或者说，先变动最左的下标，从左向右，最后变动最右的下标。

如对三维数组 A[2..4, 0..1, 1..3]，按行存储的元素次序为：

$a_{201}, a_{202}, a_{203}, a_{211}, a_{212}, a_{213}, a_{301}, a_{302}, a_{303}, a_{311}, a_{312}, a_{313}, a_{401}, a_{402}, a_{403}, a_{411}, a_{412}, a_{413}$

以三维数组 A[1..m, 1..n, 1..p]为例，按行存储时，元素 a_{ijk} 的地址为：

$LOC(i, j, k)=LOC(1, 1, 1)+[(i-1)\times n\times p+(j-1)\times p+(k-1)]\times c$

易见，不论是按行存储还是按列存储，数组元素的地址都是其下标的线性函数，所以，任一元素都可在相同的（逻辑）时间内存取，故数组是一种随机存取结构。

4.3 矩阵的压缩存储

矩阵是一种常用的数学对象，当其元素类型相同时，我们一般用二维数组来表示，此时需要存储全部的元素。但是，如果矩阵的非零元素呈某种规律分布，或者有大量的零元

① 本书对多维数组使用了两种写法（不涉及具体语言时），如 A[s..t, q..r]也写做 A[s..t][q..r]，相应地元素 a_{ij} 就分别写做 A[i,j]和 A[i][j]。但对特定的程序语言，数组的写法是有规定的，如 FORTRAN 数组 A(2:5, 7:9)，B(4,3)、C 数组 A[4][3]等。

素，此时就没有必要多次重复存储相同的非零元素或零元素了，如对相同的非零元素只分配一个存储空间，对零元素不分配空间，从而节省存储空间。这就是矩阵的**压缩存储**。

非零元素或零元素分布具有一定规律的矩阵称为**特殊矩阵**，非零元素个数很少（远远少于矩阵元素总数）的矩阵称为**稀疏矩阵**。显然，稀疏矩阵中有大量的零元素。

矩阵和数组元素的下标可以从 1 开始，也可以从 0 开始，甚至从其他某个整数开始，由于分析方法类似，就不一一加以讨论了，以下仅讨论下标从 1 开始的情况。

4.3.1　特殊矩阵

1．对称矩阵

若一个 n 阶方阵 A[1..n][1..n]的元素满足：

$a_{ij}=a_{ji}$　　　　$1\leqslant i, j\leqslant n$

则称其为 n 阶对称矩阵。例如图 4.2 就是一个 5 阶的对称矩阵。

对称矩阵中的元素关于主对角线对称，故可只存储上三角或下三角中的元素，即让每两个对称的元素共享一个存储空间。这样可节约一半左右的存储空间。

对上三角或下三角部分，又可按行或按列存储，故对称矩阵一般有 4 种不同的存储方式。不失一般性，我们讨论按行存储下三角部分，其元素存放次序如图 4.3 所示。

$$\begin{bmatrix} 1 & 4 & 6 & 1 & 2 \\ 4 & 7 & 0 & 8 & 3 \\ 6 & 0 & 5 & 4 & 0 \\ 1 & 8 & 4 & 1 & 9 \\ 2 & 3 & 0 & 9 & 6 \end{bmatrix}$$

图 4.2　对称矩阵

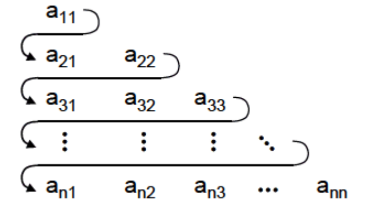


图 4.3　对称矩阵按行存放

在下三角矩阵中，第 i 行有 i 个元素，故元素总数为 1+2+…+n=n(n+1)/2。因此，我们可以用一个向量 V[1..n(n+1)/2]来存储下三角矩阵的所有元素。在这种存储方式下，为了访问原矩阵的元素 a_{ij}，必须在 a_{ij} 和 V[k]之间找到对应关系。

显然，若 $i\geqslant j$，则 a_{ij} 在下三角矩阵中。a_{ij} 之前有 i−1 行，其中有 1+2+…+(i−1)=i(i−1)/2 个元素，而 a_{ij} 是第 i 行上的第 j 个元素，因此 k=i(i−1)/2+j。

若 i<j，则 a_{ij} 在上三角矩阵中，由于 $a_{ij}=a_{ji}$，只要交换上述对应关系式中的 i 和 j 即可。所以 k 和 i、j 的对应关系为：

$$k=\begin{cases} i(i-1)/2+j & i\geqslant j \\ j(j-1)/2+i & i<j \end{cases}$$

反之，对所有的 k=1, 2,…, n(n+1)/2，也能确定 V[k]在原矩阵中对应的位置(i, j)（见习题 4.4）。即 a_{ij} 和 V[k]之间是一一对应的关系。这样，我们称向量 V[1..n(n+1)/2]为原对称矩阵的压缩存储结构，参见图 4.4。它将原来 n^2 个元素“压缩”到了 n(n+1)/2 个元素的一维数组中。

最后，如果要求元素 a_{ij} 的地址，则为：

LOC(i, j)=LOC(k)=LOC(1)+(k−1)×c

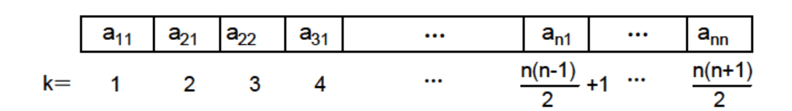


图 4.4 对称矩阵的压缩存储

2. 三角矩阵

若矩阵 A[1..n][1..n]主对线上方（或下方）的元素全部相同，均为常数 c，则称该矩阵为下三角矩阵（或上三角矩阵），见图 4.5。在大多数情况下，常数 c 为零。

$$\begin{bmatrix} a_{11} & c & \cdots & c \\ a_{21} & a_{22} & \cdots & c \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \qquad \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ c & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c & c & \cdots & a_{nn} \end{bmatrix}$$

（a）下三角矩阵　　（b）上三角矩阵

图 4.5 三角矩阵

显然，重复的常数 c 可只存储一次，其余元素有 n(n+1)/2 个，故可将三角矩阵存储到向量 V[1..n(n+1)/2+1]中，其中常数 c 存放在向量的最后一个分量中。这样，可直接利用前面对称矩阵的结果。以下三角矩阵的按行存储为例，元素 a_{ij} 和 V[k]之间的对应关系为：

$$k=\begin{cases} i(i-1)/2+j & i \geqslant j \\ n(n+1)/2\ +1 & i < j \end{cases}$$

3. 对角矩阵

对角矩阵是指，除了主对角线及其邻近的上下若干条次对角线上的元素外，其他元素均为零。在对角矩阵中，非零元素都集中在以主对角线为中心的带状区域中（对角矩阵是带状矩阵中的一种）。图 4.6（a）就是一个三对角矩阵。

如果将对角矩阵的首、末行各补上一个虚元素，则矩阵每行的非零元素个数就都相同，于是很自然地可将对角矩阵压缩到一个二维数组 $B_{n\times w}$ 中，这里 w 为对角线总条数（带宽）。对图 4.6（a）的三对角矩阵，它的二维压缩结构见图 4.6（b），元素 $b_{i'j'}$ 和 a_{ij} 的对应关系为：

$$\begin{cases} i'=i \\ j'=j-i+2 \end{cases} \qquad 1 \leqslant i,j \leqslant n$$

二维数组还可进一步转化为一维数组。当然，也可直接在原对角矩阵上，将各非零元素按行、按列或按对角线的次序存储放到一维数组中，见图 4.6（c）。

例 4.1 设有 3 对角矩阵 A[1..n, 1..n]，现将其 3 条对角线上的元素按行存于一维数组 B[1..3n−2]中，使得 B[k]=A[i, j]，见图 4.6（c）。求：

（1）用 i、j 表示 k 的下标变换公式。

（2）用 k 表示 i、j 的下标变换公式。

解：注意这里没有补充虚元素。对元素 a_{ij}，它之前有 i−1 行，非零元素个数为 3*(i−1)−1（第一行只有 2 个非零元素）；在第 i 行，对角线上的元素为 a_{ii}，由于是 3 对角矩阵，故第一个非零元素为 $a_{i,\ i-1}$，于是在第 i 行上从 $a_{i,\ i-1}$ 到 a_{ij}，元素个数为 j−(i−1)+1 个，所以到元素 a_{ij} 为止矩阵中非零元素总数为：

k=[3(i−1)−1]+[j−(i−1)+1]=2i+j−2

$$A_{n\times n}=\begin{bmatrix} a_{11} & a_{12} & 0 & \cdots & 0 & 0 \\ a_{21} & a_{22} & a_{23} & \cdots & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\ 0 & 0 & 0 & 0 & a_{n,n-1} & a_{n,n} \end{bmatrix}$$

（a）三对角矩阵

$$B_{n\times 3}=\begin{bmatrix} x & a_{11} & a_{12} \\ a_{21} & a_{22} & a_{23} \\ a_{32} & a_{33} & a_{34} \\ \vdots & \vdots & \vdots \\ a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\ a_{n,n-1} & a_{n,n} & x \end{bmatrix}$$

（b）二维压缩存储

a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{32}	a_{33}	…	$a_{n,n-1}$	a_{nn}
k= 1	2	3	4	5	6	7	…		3n−2

（c）一维压缩存储

图 4.6　三对角矩阵及其压缩存储

但反过来，已知 k，不能直接从上式求出 i 和 j，因为这里两个未知数只有一个方程，还要利用其他潜在条件（如 i、j、k 为整数等）。注意到对三对角矩阵的每一行 i，列号 j 的取值只能是 i−1、i、i+1，即 $i-1\leqslant j\leqslant i+1$，所以从上式得：

$2i+(i-1)-2\leqslant k\leqslant 2i+(i+1)-2$，即 $3i-3\leqslant k\leqslant 3i-1$，从而 $(k+1)/3\leqslant i\leqslant (k+3)/3$。

由该式可得 $k/3<i\leqslant k/3+1$，由于 i 为整数，所以 $i=\lfloor k/3\rfloor+1$[①]。

由上式还可得 $(k+1)/3\leqslant i<(k+1)/3+1$，所以 $i=\lceil (k+1)/3\rceil$。不难发现，这两者是相等的。

求出 i 后，将它带入上面 k 的表达式，便可得：

$j=k-2i+2=k-2\lfloor k/3\rfloor$，或 $j=k+2-2\lceil (k+1)/3\rceil$。

上述的几种特殊矩阵，其非零元素的分布有规律可循，总能找到一种方法将它们压缩存储到一个向量中，并且能找到矩阵中的元素和该向量下标的对应关系，从而仍然可以对矩阵元素进行随机存取。

4.3.2　稀疏矩阵

在存储稀疏矩阵时，为了节省存储单元，很自然的方法是只存储非零元素。但非零元素的分布一般是没有规律的，元素的存储次序无法反映它们之间的逻辑关系，所以必须显式地存储每个元素的行列逻辑次序。最简单的方法是将非零元素的值和它所在的行号、列号作为一个结点存放在一起，于是矩阵的每一个非零元素就由一个三元组（行号，列号，元素值）唯一确定。显然，稀疏矩阵的压缩存储会失去随机存取功能。

所有非零元素对应的三元组构成的集合就是稀疏矩阵的逻辑表示，它有两种常用的存储方式：三元组表与十字链表。

1．三元组表

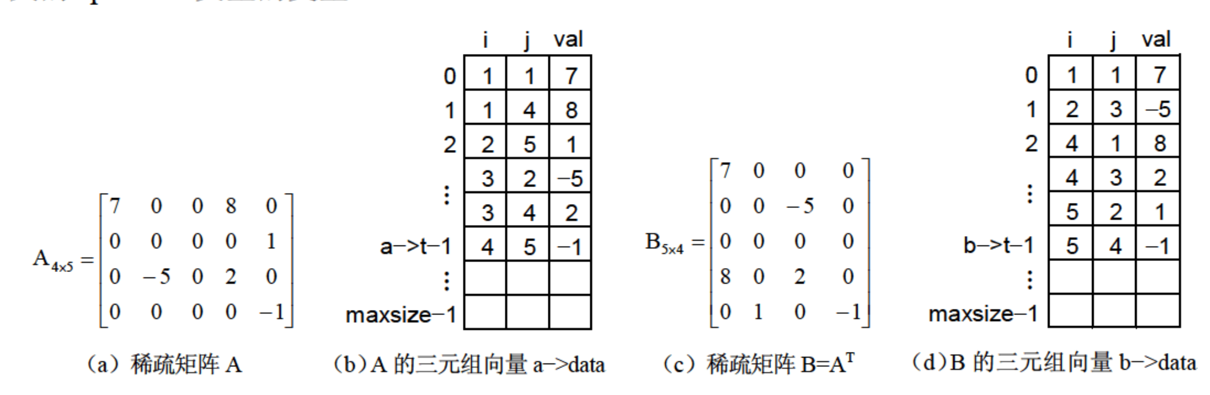
将稀疏矩阵非零元素的三元组按行序（或列序）的顺序排列，则得到一个结点均是三元组的线性表。该线性表的顺序存储结构称为稀疏矩阵的**三元组表**。因此，三元组表是稀疏矩阵的一种顺序存储结构。在以下讨论中，假定三元组按行序排列。

① $\lfloor a\rfloor$表示小于等于 a 的最大整数，即对 a 取下整数，$\lceil a\rceil$表示大于等于 a 的最小整数，即对 a 取上整数，比如$\lfloor 6.x\rfloor=6$，$\lceil 6.x\rceil=7$。

与顺序表类似，在三元组表中一般还要指出当前表的长度，即非零元素的个数，这也正是稀疏矩阵运算中经常需要的。显然，要唯一确定稀疏矩阵，还必须知道矩阵的行数和列数。三元组表的类型说明如下：

```
typedef int datatype;        //矩阵元素的数据类型，假设为 int
const int maxsize=100;       //非零元素个数的上限，三元组表的容量
typedef struct {
  int i,j;                   //行号、列号
  datatype val;              //元素值
} nodetype;                  //三元组结点类型
typedef struct {
  int m,n;                   //行数、列数
  int t;                     //当前表长，即非零元素个数
  nodetype data[maxsize];    //三元组表
} spmatrix;                  //稀疏矩阵类型
```

例如，图 4.7（a）的稀疏矩阵 A 对应的三元组表如图 4.7（b）所示，其中 a 是以上定义的 spmatrix 类型的变量。



$$A_{4\times5}=\begin{bmatrix}7&0&0&8&0\\0&0&0&0&1\\0&-5&0&2&0\\0&0&0&0&-1\end{bmatrix}\qquad B_{5\times4}=\begin{bmatrix}7&0&0&0\\0&0&-5&0\\0&0&0&0\\8&0&2&0\\0&1&0&-1\end{bmatrix}$$

（a）稀疏矩阵 A　（b）A 的三元组向量 a−>data　（c）稀疏矩阵 $B=A^T$　（d）B 的三元组向量 b−>data

图 4.7　稀疏矩阵和它的三元组表存储结构

如果考虑到 C/C++语言数组下标从 0 开始，而矩阵行、列号一般从 1 开始，我们也可从数组的 1 号单元开始存放非零元素，而 0 号单元正好可用来存放矩阵的行数、列数和非零元素个数。这时上述稀疏矩阵的类型定义以及以后的有关运算需要略作修改。

稀疏矩阵三元组表的基本运算也是读 GET(i, j)和写 SET(i, j, x)，实现时要先进行行、列号的查找（不能随机存取），而在写时，三元组表中可能没有对应元素（即该元素原来为零），这时要在三元组表中插入一个元素；如果 x=0，则要在三元组表中删除对应元素（若存在的话）。这是因为三元组表不存储零元素。有了读写运算，就可以像访问普通数组那样访问三元组数组了。其他运算可根据使用的需要设置，如矩阵的转置、加法、乘法等。下面主要讨论矩阵的转置 TRANS()在三元组表上的实现。

矩阵的转置是指将它的行列互换，如一个 m×n 的矩阵 A，转置后得到一个 n×m 的矩阵 B，则 A[i][j]=B[j][i]。例如图 4.7（a）中的矩阵 A 和图 4.7（c）中的矩阵 B 互为转置矩阵。

利用 GET 和 SET 运算，可实现矩阵的转置：

```
for(i=1;i<=m;i++)
  for(j=1;j<=n;j++)
    B.SET(j,i,A.GET(i,j));
```

这个算法虽然简单，但效率不高，其时间复杂度为 O(m×n)；而对稀疏矩阵，GET、SET 不能对元素随机存取，还要增加查找元素位置的时间开销，效率更低。为了提高效率，可不通过 GET 和 SET 而直接实现转置。

如果在转置时简单地将每个三元组的行号和列号互换，则转置后的三元组将不是按行序而是按列序排列的，这时需要对三元组重新排序，总的时间复杂度为 $O(t)+O(t\log_2 t)$。这里，$O(t\log_2 t)$是基于比较运算的排序算法的最好平均时间复杂度，在简单排序（如冒泡排序等）情况下，时间复杂度可能上升为 $O(t^2)$。

为了降低时间复杂度，显然应该设法避免单独的排序运算，这就需要在进行元素行号和列号互换的过程中，顺便按行序排列。具体来说，一般有两种算法，下面分别介绍。

（1）按列序转置，顺序存放

注意到矩阵 B 的行就是矩阵 A 的列，如果将 A 的三元组表按列序转置，则得到的 B 的三元组表必定按行序排列。这样，可对 A 的三元组从头到尾进行扫描，分别取出每一列上的非零元素，将其行列交换后，依次存入 B 的三元组中。具体算法如下：

```
void trans(spmatrix *A,spmatrix *B) {
  int pa,pb,col;
  B->m=A->n;
  B->n=A->m;
  B->t=A->t;
  if(A->t<1) return;             //没有任何非零元素
  pb=0;                          //pb 为 B 中三元组表当前空位置
  for(col=1;col<=A->n;col++)     //对 A 所有列循环
    for(pa=0;pa<A->t;pa++)       //对 A 三元组循环，查找每一列号
      if(A->data[pa].j==col) {
        B->data[pb].i=A->data[pa].j;
        B->data[pb].j=A->data[pa].i;
        B->data[pb].val=A->data[pa].val;
        pb++;
      }
}
```

由于三元组表的元素按行序排列，上述算法在扫描 A 的三元组表时，同一列上的非零元素必然是按行号大小的顺序出现的，所以转置后的三元组表中行号相同的元素正好按列号排列。显然，算法的时间复杂度为 O(n×t)。作为对比，非压缩矩阵转置的时间复杂度为 O(m×n)，而一般非零元素个数 t>>行数 m，所以上述转置算法的效率远低于非压缩矩阵的转置。

（2）按行序转置，按列索引存放

上述转置算法要反复搜索三元组表以查找同列元素，从而影响了效率。如果对矩阵 A 每一列的第一个非零元素建一个索引，指出它在 B 的三元组表中的位置，并在该列元素存放时，动态修改列索引，则任何元素都可按列索引存放，就不必按列序转置了。

这里先建立一个列索引 cpos[1..n]，其中 cpos[i]表示原矩阵第 i 列上第 1 个非零元素在 B 的三元组表中的位置。为了运算方便，再设一个数组 cnum[1..n]，其中 cnum[i]存放原矩阵中第 i 列上非零元素的个数。cpos 可用下列递推公式求得：

列号	1	2	3	4	5
列非零元数 cnum	1	1	0	2	(2)
列起始位置 cpos	0	1	2	2	4

图 4.8　矩阵 A 的 cnum 和 cpos

```
cpos[1]=0
cpos[i]=cpos[i-1]+cnum[i-1]    2≤i≤n
```

例如，对图 4.7（a）所示的矩阵 A，cnum 和 cpos 的值如图 4.8 所示（最后一列的非零元素个数不必求出，未用到）。具体算法如下：

```
void trans2(spmatrix *A,spmatrix *B) {
  int pa,pb,col;
  int *cpos,*cnum;
  B->m=A->n;
  B->n=A->m;
  B->t=A->t;
  if(A->t<1) return;                //无非零元素
  cpos=new int[A->n+1];             //动态分配辅助空间
  cnum=new int[A->n+1];
  for(col=1;col<=A->n;col++) cnum[col]=0;
  for(pa=0;pa<A->t;pa++) {
    col=A->data[pa].j;
    cnum[col]++;                    //累加每列非零元素个数
  }
  cpos[1]=0;
  for(col=2;col<=A->n;col++)
    cpos[col]=cpos[col-1]+cnum[col-1];  //递推：每列第一个非零元素在B中的位置
  for(pa=0;pa<A->t;pa++) {
    col=A->data[pa].j;
    pb=cpos[col];                   //该列在B中开始位置
    B->data[pb].i=A->data[pa].j;
    B->data[pb].j=A->data[pa].i;
    B->data[pb].val=A->data[pa].val;
    cpos[col]++;                    //该列下一个元素在B中位置
  }
  delete []cpos;                    //释放辅助数组空间
  delete []cnum;
}
```

算法的时间复杂度为 O(n+t)，比前一个效率高得多，故这种转置方法又称快速转置。

即使对非稀疏矩阵，该算法也有意义，因为 t 接近 m×n 时，时间复杂度变为 O(m×n)，与不压缩直接转置的算法相当。

顺便指出，该算法的数组 cnum 可不单独设置，而借用 cpos 的空间，即将 cnum[i]存放到 cpos[i+1]位置上，递推公式改为 cpos[i]=cpos[i−1]+cpos[i]，具体算法略。

实际上，在稀疏矩阵的三元组表示中，我们也可以建立一个附加的行索引，称为带行（索引）表的三元组表。这样可以比较方便地找到某一行的第一个非零元素以及该行非零元素的个数，其原理与上面建列索引类似。

2．十字链表

稀疏矩阵非零元素的位置和个数可能会经常发生变化，如矩阵相加 A=A+B，在 A 中就可能出现新的非零元素，或原来的非零元素变成了零元素，这就涉及结点的插入和删除运算。三元组表是一种顺序存储方法，对插入和删除运算是不合适的，因为会引起大量结点的移动，此时采用链式存储结构就比较合适。

稀疏矩阵常用的链式存储结构是十字链表。不过，十字链表的应用远不止稀疏矩阵，一切具有正交关系的结构，都可采用十字链表这种存储结构。

在十字链表中，每个结点除了存放非零元素的三元组外，还增加了行、列两个指针：

行指针用来指向本行中的下一个非零元素；列指针用来指向本列中的下一个非零元素。即通过行指针将同一行上的非零元素链接在一起，通过列指针将同一列上的非零元素链接在一起。因此，每一个非零元素 a_{ij} 既是第 i 行链表上的一个结点，又是第 j 列链表上的一个结点，就好像处在一个十字路口上，故称这样的链表为**十字链表**或正交链表。这是一种多重链表结构。

显然，如果只有行链表（或列链表）也可用来表示稀疏矩阵，这样虽可节省一定的空间，但如果要找同一列（或同一行）上的非零元素就不如十字链表方便了。

十字链表的结点结构见图 4.9（a），其中各字段的含义为：

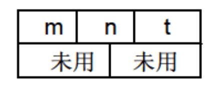
- i、j、val——非零元素的行号、列号和元素值。
- down——列指针，指向同列中下一个非零元素结点。
- right——行指针，指向同行中下一个非零元素结点。

十字链表的类型定义如下：

```
typedef int datatype;       //矩阵元素的数据类型，假设为 int
typedef struct node * pointer;  //链表结点指针类型
struct node {
  int i,j;
  datatype val;
  pointer down,right;       //列指针和行指针
};
typedef pointer xlink;      //十字链表头指针类型
```

i	j	val
down	right	

（a）非零元素结点



（b）行/列表头结点

m	n	t
未用	未用	

（c）总表头结点

图 4.9　十字链表的三种结点类型

为了运算方便，对每一个行链表和列链表都增加一个头结点，头结点的行、列域未用（但可用来存放本行或本列中的非零元素个数，见图 4.9（b））。例如，图 4.7（a）所示稀疏矩阵 A 的十字链表如图 4.10 所示。

由图 4.10 可见，每个列链表的头结点，只用列指针 down 指向本列中的第一个非零元素；每个行链表的头结点，只用行指针 right 指向本行中的第一个非零元素；这样，两组表头结点可以合用，即第 i 行链表和第 i 列链表共享一个表头结点 H_i，以节省存储空间。

所有的头结点组织成头结点数组，当下标从 1 开始使用时，数组大小为 1+max(m, n)，这时零号单元的三元组可用来存放矩阵的行数、列数、非零元素的个数等总体信息，该单元相当于总表头结点（见图 4.9（c））。

也可把所有的头结点组织成链表，这在矩阵的大小有变化时比较有利，但找某行或某列的头结点时则需要在链表中搜索。

十字链表上的基本运算，除了数组的读和写之外，还有链表本身的一些运算，如初始化、插入、删除等，这里只就运算的一些基本方法进行说明。

（1）初始化

生成 m 行 n 列矩阵的空十字链表，这时只有头结点。动态申请大小为 1+max(m, n)的头结点数组，0 号单元的三元组内存放矩阵的行数 m、列数 n、非零元素的个数 0。

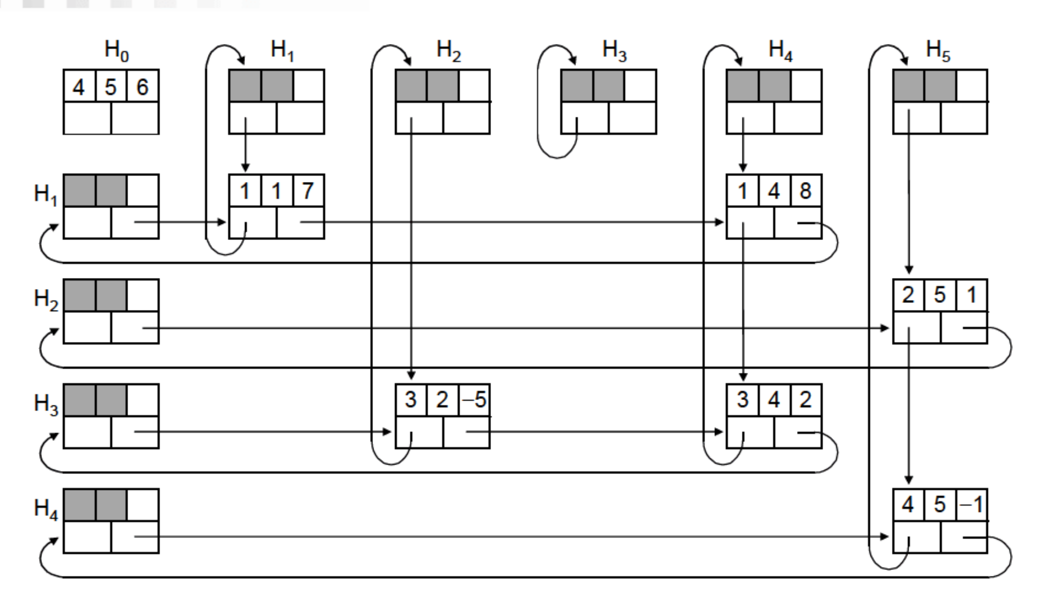


图 4.10 稀疏矩阵 A 的十字链表

（2）插入

插入新结点时，分别在相应的行、列两个链表中插入。

（3）删除

删除结点时，分别在相应的行、列两个链表中删除。

（4）读 GET

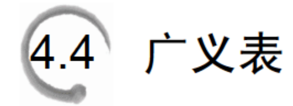
GET 运算实现按数组方式访问十字链表，即已知行号和列号求出元素值或结点的地址。方法是搜索对应的行链表，查找列号为给定列号的结点（或搜索对应的列链表，查找行号为给定行号的结点）。

（5）写 SET

SET 运算实现按数组方式给元素赋值，即已知行号和列号，给对应的元素赋值。与 GET 类似，需要先按给定的行号或列号查找对应的元素。如果查找到了，就对其赋值，但当赋零值时应删除对应的结点；若查找不到，则要插入一个新结点。

有了基本运算，就可以完成十字链表的其他运算了，如建表。显然，它由两步完成：先建一个空表，这通过初始化完成；然后依次输入各个三元组，将它们插入到十字链表中，这可通过插入算法完成。

以上三元组表、十字链表中非零元素的存储信息为三元组（行号，列号，值），也可存储为二元组（数组元素按行或按列排列时的序号，值），以后要用行号和列号时再通过排列关系计算出来。这样压缩了行、列信息的储存空间，但增加了以后使用的时间。



4.4 广义表

4.4.1 广义表的基本概念

广义表（Generalized List）又称**列表** Lists[①]，是 n（n≥0）个元素 $a_1, a_2, \cdots, a_n$ 的有限序

① 这里用复数形式以区分一般的表（list）。

列，其中 a_i 或者是原子或者是一个广义表，通常记为 LS=(a_1, a_2, …, a_n)。这是一个递归定义。若 a_i 本身也是一个广义表，则称它为 LS 的**子表**。LS 中元素 a_i 的个数（不计 a_i 的内部结构）称为 LS 的**长度**。不含任何元素（长度为 0）的表称为**空表。**

可见，若不考虑元素 a_i 的内部结构，则广义表 LS 就是一个线性表，可看作线性表的推广。反过来说，线性表就是不含子表的广义表。

对非空广义表，它的第 1 个元素称为**表头，**除去表头后其余元素构成的表称为**表尾**。显然，表尾一定是子表，但表头可以是原子，也可以是子表。

将广义表展开后所含括号的最大嵌套层数称为**深度**。

广义表的成员可以为子表，子表的成员又可以是子表，……，这使得广义表呈现出多层次性和多样性。如果允许广义表的某成员内含有广义表自己，则称此广义表为**递归表**。在广义表中，某个成员（原子或子表）可能出现多次，但每次出现应代表的是同一个目标，或者说是对同一目标的共享。允许元素共享的广义表称为**再入表**（Reentrant List）。如果表中没有共享和递归的成分，即没有任何成分出现多次，则称此广义表为**纯表**（Pure List）。各种表之间的关系为：

线性表 ⊂ 纯表 ⊂ 再入表 ⊂ 递归表

在书写上，为了区分原子和广义表，一般用大写字母表示广义表，用小写字母代表原子。下面给出几个广义表的例子。

A=()：空表，无表头，无表尾，长度为 0，深度为 1。注意，空表无表头也无表尾。

B=(A)=(( ))：表头为( )，表尾为( )，长度为 1，深度为 2。这里表头和表尾都为空表。

C=(a, b)：表头为 a，表尾为(b)，长度为 2，深度为 1。这是个线性表。

D=(C, x)=((a, b), x)：表头为(a, b)，表尾为(x)，长度为 2，深度为 2。

E=(y, D)=(y, (C, x))=(y, ((a, b), x))：表头为 y，表尾为(D)，长度为 2，深度为 3。

F=(C, D)=((a, b), ((a, b), x))：表头为 C，表尾为(D)，长度为 2，深度为 3。这是再入表。

G=(z, G))=(z, (z, (z, (…))))：表头为 z，表尾为(G)，长度为 2，深度为∞。这是递归表。

有时，为了强调广义表名称，可将表名写在表的左括号前面，如上面各表可写为：

A( )

B(A( ))

C(a, b)

D(C(a, b), x)

E(y, D(C(a, b), x))

F(C(a, b), D(C(a, b), x))

G(z, G(z, G(…)))

广义表可用图形形象地表示，图 4.11 给出了上面几个广义表的图形表示，其中分支结点对应广义表，非分支结点（即叶子）对应原子或空表。如果与以后我们将要介绍的树、图等内容联系起来，不难看到，没有共享和递归成分的纯表（a）～（e）相当于树形结构；有共享成分的再入表（f）相当于有向无环图（DAG）；有递归成分的递归表（g）相当于有回路的有向图。

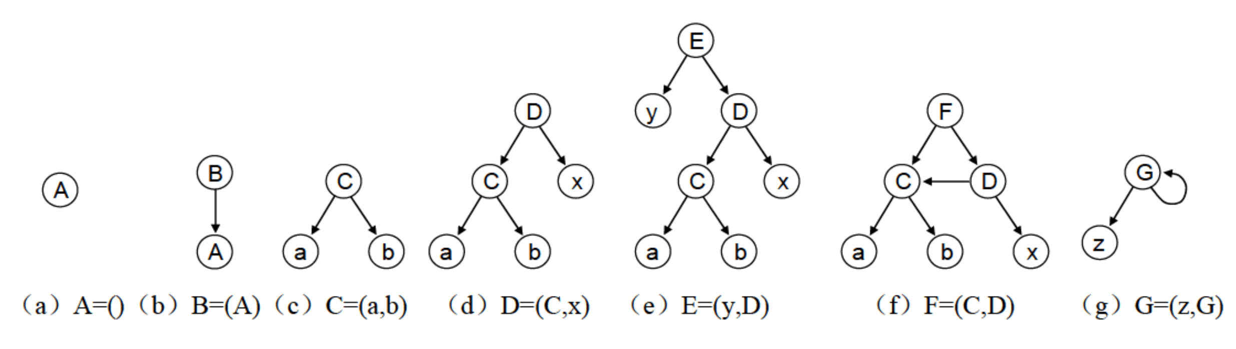


图 4.11　广义表的图形表示

这样看来，广义表不仅是线性表的推广，也是树的推广。由于广义表的元素可以递归，使得广义表具有很强的表达能力，这是广义表最重要的特性。

广义表的基本运算，除包括线性表的基本运算外，还有求深度、求表头、求表尾、求成员、遍历等。这些运算中大部分与对应的线性表、树或图的运算类似，只有求表头和表尾是广义表特有的运算。著名的人工智能语言 LISP 就是以广义表为数据结构的，其中就连程序也表示为一系列的广义表，通过求表头和表尾来实现有关运算。

例 4.2　试通过取表头 head(LS)和取表尾 tail(LS)运算，从广义表 A=(x, (a, b), y)中取出原子 b。

解：在广义表中取某个元素，需要将该元素所在的子表逐步分离出来，直到所求的元素成为某个子表的表头，再用取表头运算取出。注意，最终取出某个元素时，不能是取表尾，因为它得到的是该元素组成的子表，而不是元素本身。本例的运算过程为：

（1）取表尾 tail(A)：得到 B=((a, b), y)。

（2）取表头 head(B)：得到 C=(a, b)。

（3）取表尾 tail(C)：得到 D=(b)。

（4）取表头 head(D)：得到 b。

总的过程为：head(tail(head(tail(A))))。

4.4.2　广义表的储存结构

广义表的成员可以是原子，也可以是子表，它们结构不同，难以用顺序存储结构表示，一般采用链式存储结构，其中有两种结构的结点：表结点和原子结点，它们一般通过设标志位来区分。下面介绍广义表的两种链式存储结构。

1．头尾链表

基本思想是把广义表不断分成表头和表尾。表结点包含三个域：标志域、表头指针域和表尾指针域；原子结点包含两个域：标志域和值域。结点结构为：

表结点：

tag=1	head	tail

原子结点：

tag=0	data

设广义表 G=(a, ( ), ((b, c), d))，其头尾链表表示如图 4.12 所示。

2．扩展线性链表

基本思想是把广义表看成由若干元素组成的“线性表”，其中若某元素为子表，则再

建立该子表的“线性表”。所有结点都包含三个域：标志域、值域或表头指针域、后继指针域，具体结点结构为：

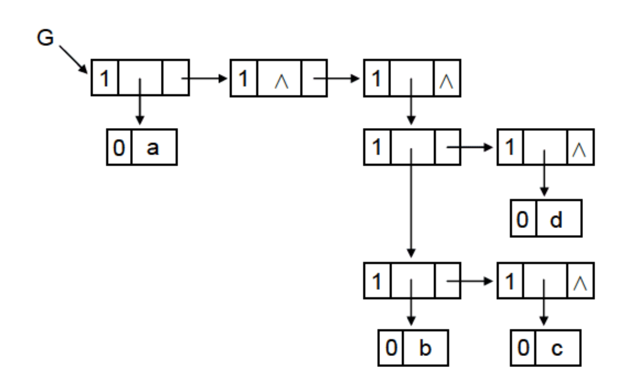


图 4.12　广义表的头尾链表表示示意图

表结点： | tag=1 | head | next |

原子结点： | tag=0 | data | next |

仍设广义表 G=(a, ( ), ((b, c), d))，其扩展线性链表表示如图 4.13 所示。

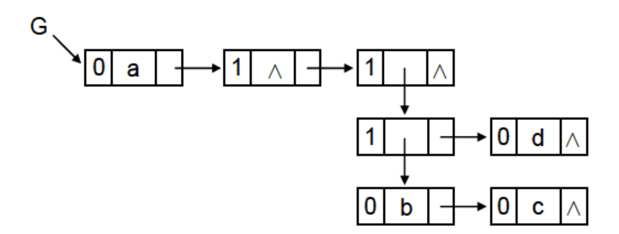


图 4.13　广义表的扩展线性链表表示示意图

有了储存结构，就可实现具体的运算。以扩展线性链表为例，（1）求长度时，就是求相应的同层结点所构成的单链表的长度；（2）求深度时，就是各元素“深度”中的最大值+1，元素的深度定义是：若其为空表则为 1，若为原子则为 0，否则为子表，则递归处理；（3）建表时，先建立表头指针 L，然后根据输入的字符：左括号、逗号、字母、空表符（如约定为“；”），右括号、输入结束符（如约定为“#”）等，确定属于子表、原子、空表还是结束标志，然后决定是否建立子表的表头指针等；（4）遍历时，沿单链表搜索，若是原子则访问结点（如输出内容），若是子表，则递归处理。具体算法略。

习　题　四

4.1　设有三维数组 A[−1..0][1..2][3..4]，请分别按行序和列序列出各元素。

4.2　对一般三维数组 A[r1..r2][s1..s2][t1..t2]，分别计算元素 A[r][s][t]的按行存储与按列存储的地址（设每个元素占 c 个单元）。

4.3　设上三角矩阵 A[1..n][1..n]按行存放在数组 B[1..m]（m 足够大），使得 B[k]=A[i][j]。试推导 k 和 i、j 的关系。

4.4 设对称矩阵 A[1..n][1..n]按行序将下三角存储在一维数组 B[1..n(n+1)/2]中，试推导元素 B[k]在原矩阵 A 中对应的行号与列号。

4.5 设稀疏矩阵 A 以三元组表存放，编一算法求元素 A[i][j]。

4.6 设 A、B 两个稀疏矩阵用三元组表表示，编写算法，计算 A×B、A+B。

4.7 设 A、B 两个对称矩阵按行存储下三角，编写算法，计算 A×B、A+B。

4.8 设 A、B 两个普通矩阵按行存储，编写算法，按一维数组寻址公式计算 A×B、A+B。

4.9 写一个将用十字链表存储的稀疏矩阵转置的程序。

4.10 通过取表头 head()和取表尾 tail()运算，取出下表的 here：

L=(this, (there, (the, here), where), which, what)

4.11 写出上题广义表的长度、深度、表头、表尾。

第5章 树形结构

树形结构是一种十分重要的非线性结构，可描述数据元素间一对多的逻辑关系，其结点之间形成分支和层次关系，类似于自然界中的树。在客观世界中，有许多事物本身就呈现树形结构，如家族关系、部门机构设置等，用树来表示就非常简便，也非常形象和自然。另外，有些算法，也常常要借助树形结构来解决。

本章介绍树形结构中树、森林、二叉树等的基本内容①并列举一些简单应用，重点是二叉树。在后面几章中，还会涉及到树和二叉树的一些其他应用。

5.1 树的概念

在现实世界中，有很多问题可用树形结构来描述。如一个零部件的组成就可以很自然地表示成一个树形结构。图 5.1 表示的就是某鼠标器的组成：它由电路和机械两大部分组成；前者由接口、按键检测、滚轮检测和主电路组成；后者由滚轮、按键、壳体等组成。这类图形看上去就像一棵倒画的树，“树形结构”即由此得名。

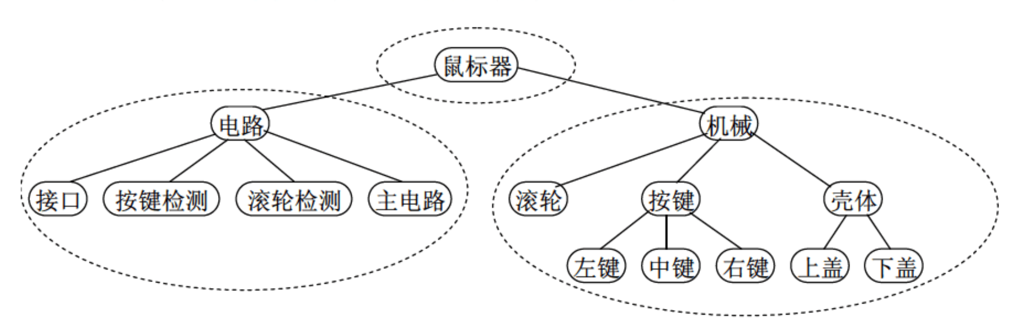


图 5.1　鼠标器的组成示意图

图 5.1 可分解为三个部分：鼠标器本身、电路组成、机械组成，如图中虚线所示。而电路组成和机械组成还可进行类似的分解。分解出的小组成部分仍然保持着树形结构，直到每部分只有一个结点为止。这一特点是一切树形结构都具备的。从所有类似的问题可抽象出树的递归定义。

① 树形结构中的一些术语或名称在不同文献中可能有较大差异，如同一个概念可能名称不同，而同一个名称也可能含义不同，需要注意。

树（Tree）是 n（n≥0[①]）个结点的有限集合 T，它或者为空（n=0），或者满足以下条件：

（1）有且仅有一个特定的称为**根**（Root）的结点。

（2）其余结点分为 m（m≥0）个互不相交的子集 $T_1, T_2, \cdots, T_m$，其中每个子集又是一棵树，称其为根的**子树**（Subtree）。

从该定义知，只有一个结点的集合是一棵树，该结点就是根，它无子树。如果集合中元素个数大于 1，则它至少含有一棵子树。

在树的树形图表示中，结点一般用圆圈表示，结点的名字写在圆圈旁边，有时也写在圆圈内。除了树形表示法外，在不同的应用场合，树还可有其他表示法。如图 5.2（a）所示的树还可用图 5.2（b）、（c）和（d）来表示，其中（b）用的是凹入表示法，类似于书的目录，适合文本模式下树的屏幕显示和打印输出；（c）采用的是嵌套集合表示法，利用集合的包含关系来描述，树根所在的集合最大；（d）采用的是广义表表示法。

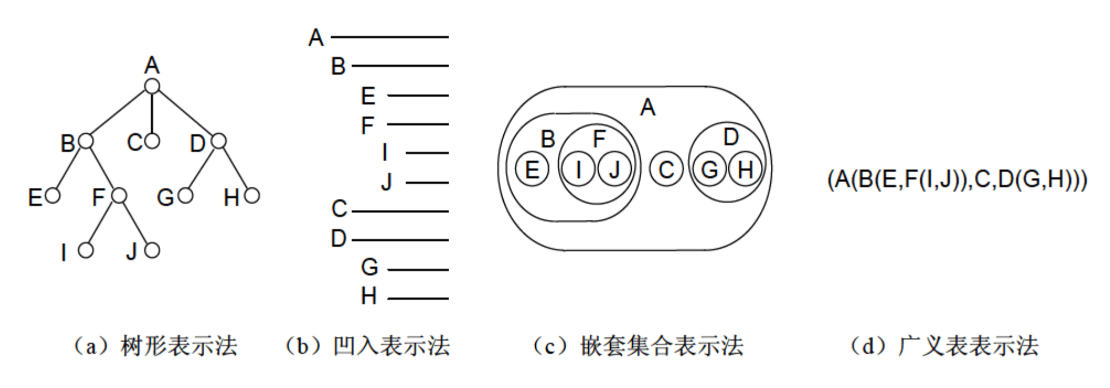


图 5.2　树形结构的表示示意图

另外，在日常工作中表示事物的分类和组成时，也常将图 5.2（a）的树形表示法转动 90°使用，例如图 5.3 对 C/C++语言数据类型的分类表示。

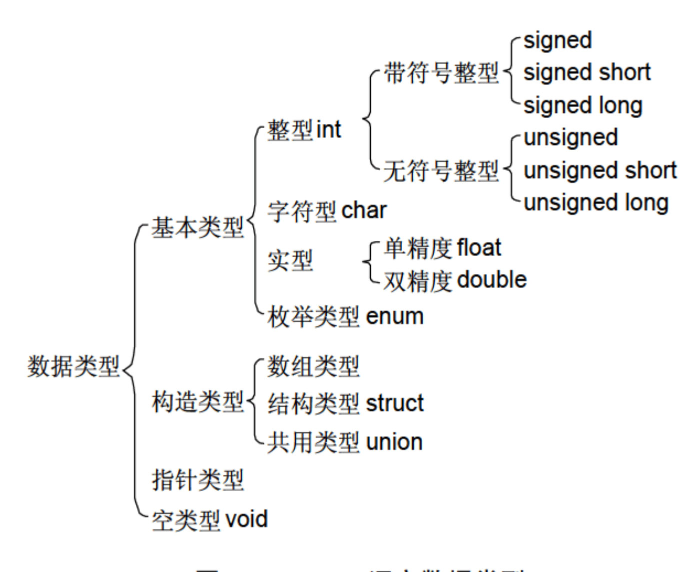


图 5.3　C/C++语言数据类型

① 有的文献只允许 n>0，即没有空树的概念。

一个结点的子树个数称为该结点的**度**（Degree）。一棵树的度是指该树中结点的最大度数。度为 k 的树也称为 k 叉树，它的每个结点最多有 k 个子树。度为零的结点称为**叶子**（Leaf）、叶结点或终端结点。度不为零的结点称为分支结点或非终端结点，除根结点之外的分支结点统称为**内部结点**，根结点也称为**开始结点**。

树中某个结点的子树之根称为该结点的**孩子**（Child）或儿子，相应地，该结点称为孩子的**双亲**（Parents）或父亲。同双亲的孩子互称为**兄弟**（Brother 或 Sibling）。双亲是兄弟关系的结点称为**堂兄弟**。

若树中存在一个结点序列$\{k_1, k_2, \cdots, k_j\}$，使得$k_i$是$k_{i+1}$的双亲，则称这个结点序列为从$k_1$到$k_j$的一条**路径**（Path）或道路，或称从$k_1$到$k_j$有路径。路径中边（即连接两个结点的线段）的个数称为路径长度。路径中的结点序列“自上而下”地通过路径上的各边。若树中结点 k 到k_s有路径，则称 k 是k_s的**祖先**（Ancestor），k_s是 k 的**子孙**（Descendant）。

结点的**层数**（Level）是从根开始算起的，根为第 1 层[①]，其他结点的层数为其双亲的层数加 1。树中结点的最大层数，称为树的**高度**（Height）或**深度**（Depth）。

若树中每个结点的各子树从左到右是有次序的（即位置不能互换，否则互换后认为是不同的树），则称该树为**有序树**（Ordered Tree）；否则称为**无序树**（Unordered Tree）。

若两棵树中，各结点对应相等，对应结点的相关关系也对应相等，则称这两棵树**相等**（等价）；若两棵树中，适当地重命名其中一棵中的结点，可以使两者相等，则称这两棵树**同构**。

森林（Forest）是 m（m⩾0）棵互不相交的树的集合。显然，删去一棵树的根，就得到一个森林；反之，加上一个结点作树根，森林就变为一棵树。

树形结构的逻辑特征可用结点之间的父子关系来描述：树中任一结点都可有零个或多个直接后继（即孩子），但最多只能有一个直接前趋（即双亲）。树中只有根无前趋（故称开始结点），叶子无后继（故称终端结点）。显然，父子关系是非线性的。祖先与子孙的关系是父子关系的延伸。

树的基本运算有以下 6 种。

（1）初始化 INITIATE(T)：置 T 为空树。

（2）求双亲 PARENT(T, x)：求树 T 中结点 x 的双亲结点。若结点 x 是树 T 的根结点或结点 x 不在树 T 中，则函数值为“空”。

（3）求孩子 CHILD(T, x, i)：求树 T 中结点 x 的第 i 个孩子结点。若结点 x 是树 T 的叶子或无第 i 个孩子或结点 x 不在树 T 中，则函数值为“空”。

（4）插枝 INSERT(T, x, i, Y)：将以结点 Y 为根的树置为树 T 中结点 x 的第 i 棵子树。若原树 T 中无结点 x 或结点 x 的子树个数<i−1，则空操作。

（5）剪枝 DELETE(T, x, i)：删除树 T 中结点 x 的第 i 棵子树。若树 T 中无结点 x 或结点 x 的子树个数少于 i，则空操作。

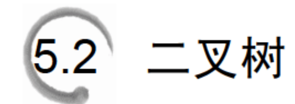
（6）遍历 TRAVERSE(T)：按某种次序对树中每个结点访问一次且仅访问一次。

在实际应用中，可根据需要对这些基本运算进行适当增减，如增加求根、求右兄弟、

---

① 有的文献层数从 0 开始计算，这时深度和/或高度仍指最大层数，或最大层数+1。

建树、查找和删除结点等运算。



5.2 二叉树

二叉树是树形结构的一个重要类型，它的存储结构和算法都比较简便，特别适合于计算机处理。即使一般形式的树也可简单地转换为二叉树，再进行相应的处理。所以二叉树显得特别重要。

5.2.1 二叉树的概念

二叉树（Binary Tree）是 n（n≥0）个结点的有限集，它或者为空（n=0），或者由一个根结点及两棵互不相交的、分别称作该根的左子树和右子树的二叉树组成。这是个递归定义。由于二叉树本身及子二叉树都可为空，故二叉树有 5 种形态，如图 5.4 所示。

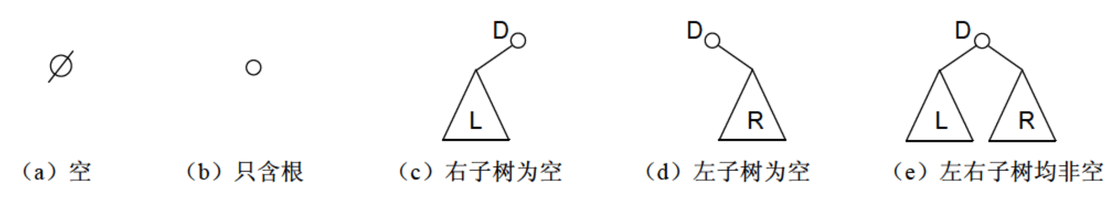


图 5.4 二叉树的 5 种基本形态

在二叉树中，每个结点最多只能有两棵子树，并且有左右之分，显然它不是无序树。但它与度为 2 的有序树也不同。因为有序树的孩子之间虽然有左右之分，但若只有一个孩子，则不分左右；而二叉树即使只有一个孩子，也要严格区分左右。可见，二叉树既不是树的特殊情形，也不是有序树的特殊情形。

例如，图 5.5 是三棵不同的二叉树，它们与图 5.6 的普通树（有序或无序）相似，但不等同。如果将这 4 棵树都看成普通树，则它们就相同了。

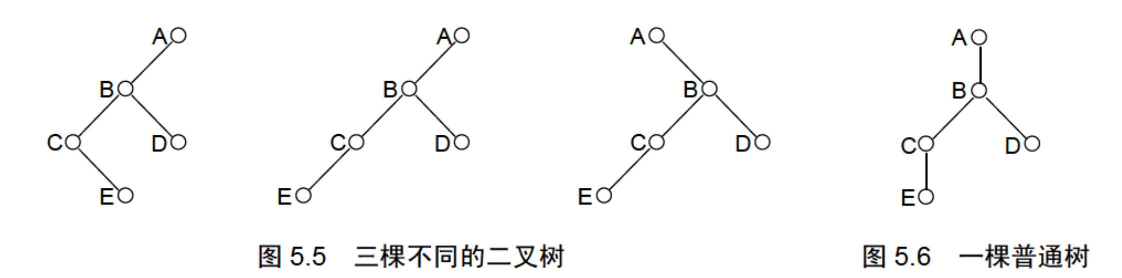


图 5.5 三棵不同的二叉树　　图 5.6 一棵普通树

和树类似，二叉树的基本运算有以下 6 种。

（1）初始化 INITIATE(BT)：加工型运算，建立一棵空二叉树 BT。

（2）求双亲 PARENT(BT, x)：引用型运算，求二叉树 BT 中结点 x 的双亲。若结点 x 是二叉树 BT 的根或二叉树 BT 中无此结点，则函数值为“空”。

（3）求左孩子 LCHILD(BT, x)及右孩子 RCHILD(BT, x)：引用型运算，分别求二叉树 BT 中结点 x 的左孩子及右孩子。若 x 为叶子或 x 不在二叉树 BT 中，则函数值为“空”。

（4）插枝 INSERTLEFT(BT, x, Y)及 INSERTRIGHT(BT, x, Y)：加工型运算，分别将以结点 Y 为根的二叉树置为二叉树 BT 中结点 x 的左子树和右子树。若原树 BT 中无结点 x 或结点 x 已有相应的子树，则空操作。

（5）剪枝 DELLEFT(BT, x)及 DELRIGHT(BT, x)：加工型运算，分别删除二叉树 BT 中结点 x 的左子树和右子树。若 x 无左子树或右子树，或 x 不在 BT 中，则为空操作。

（6）遍历 TRAVERSE(BT)：引用型运算，按某种次序访问二叉树中各个结点，使每个结点只被访问一次。

这些基本运算在实际应用中也可根据需要适当增减，如增加求根、求左右兄弟、建树、查找和删除结点等运算。

5.2.2　二叉树的性质

性质 1　二叉树第 i 层上的结点数最多为 2^{i-1}（i≥1）。

证：使用归纳法。i=1 时，结论显然成立。设 i=k−1 时结论亦成立，即第 k−1 层上的结点数最多为 2^{k-2}，则考虑 i=k 时的情形。由于二叉树每个结点最多有两个孩子，故第 k 层上的结点数最多是第 k−1 层上结点数的 2 倍，即最多 $2\times2^{k-2}=2^{k-1}$ 个，故命题成立。

性质 2　深度为 k 的二叉树至多有 2^k-1 个结点（k≥1）。

证：在深度相同的二叉树中，仅当每一层的结点数都达到最多时，树中结点总数才最多，于是由性质 1 可知，深度为 k 的二叉树的结点数最多为 $2^0+2^1+\cdots+2^{k-1}=2^k-1$。证毕。

如果深度为 k 的二叉树有 2^k-1 个结点，则称此二叉树为**满二叉树**①（Full Binary Tree）。它的特点是每一层上的结点数都达到了最大，即对给定的高度，它是具有最多结点数的二叉树。满二叉树中不存在度为 1 的结点，每个分支结点均有两棵高度相同的子树，且所有叶子都在最下一层上，见图 5.7（a）。

若一棵二叉树至多只有最下面两层上的结点的度可以小于 2，并且最下层上的结点都集中在该层最左边的若干位置上，则此二叉树称为**完全二叉树**②（Complete Binary Tree）。它相当于在满二叉树的最底层，从右向左连续去掉若干个结点后得到的二叉树，见图 5.7（b）。显然，在完全二叉树中，若一个结点没有左孩子，则它一定没有右孩子，即必定是叶子。

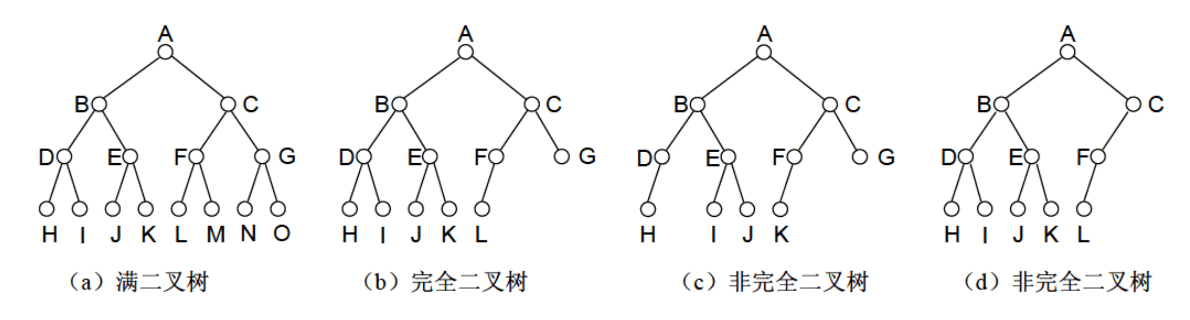


图 5.7　完全二叉树和非完全二叉树示例

① 有的文献称此为完美二叉树（Perfect Binary Tree），而满二叉树指本书所称的严格二叉树；也有的文献称此为 Complete Binary Tree，而对应本书完全二叉树的是 Nearly Complete Binary。

② 有的文献称此为顺序二叉树：与同高度的满二叉树前 n 个结点按层次顺序一一对应；也有的文献完全二叉树指本书所称的严格二叉树。

满二叉树是完全二叉树，但完全二叉树不一定是满二叉树。空二叉树和只含一个结点的二叉树既是满二叉树，也是完全二叉树。

性质 3 二叉树中，度为 0 的结点数 n_0 和度为 2 的结点数 n_2 满足 $n_0=n_2+1$。

证：因为二叉树中所有结点的度只可能是 0、1 或 2，所以结点总数 n 应等于 0 度结点数 n_0、1 度结点数 n_1 和 2 度结点数 n_2 之和：

$$n=n_0+n_1+n_2$$

另一方面，度为 1 的结点有一个孩子，度为 2 的结点有两个孩子，故二叉树中孩子结点数为 n_1+2n_2，但根不是任何结点的孩子，故二叉树中的结点总数又可表示为孩子结点数与根之和：

$$n=n_1+2n_2+1$$

结合上面两式，即可导出 $n_0=n_2+1$。

性质 4 具有 n 个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor+1$ 或$\lceil \log_2(n+1) \rceil$。

证：设 k 为所求完全二叉树的深度。由完全二叉树定义知道，它的前 k−1 层是深度为 k−1 的满二叉树，结点数为 $2^{k-1}-1$。但第 k 层上还有若干个结点，则总结点数 $n>2^{k-1}-1$。另由性质 2 知道 $n\leqslant 2^k-1$，所以：

$$2^{k-1}-1<n\leqslant 2^k-1$$

由该式得 $2^{k-1}<n+1\leqslant 2^k$，取对数后有 $k-1<\log_2(n+1)\leqslant k$，即 $\log_2(n+1)\leqslant k<\log_2(n+1)+1$，由于 k 为整数，即得 $k=\lceil \log_2(n+1) \rceil$。

另外，注意到 n 为整数，由上式还可得 $2^{k-1}\leqslant n<2^k$，取对数后有 $k-1\leqslant \log_2 n<k$，即 $\log_2 n<k\leqslant \log_2 n+1$，由于 k 为整数，又可得 $k=\lfloor \log_2 n \rfloor+1$。证毕。

对二叉树的所有结点，我们可按一定规则对其编号，其中很自然的方法是按层次顺序进行，即**层序编号**。具体说，就是从根结点开始，从上层到下层，每一层从左到右，依次给所有结点标记 1, 2, …, n，其中根的编号为 1，n 为结点总数，见图 5.8 所示。

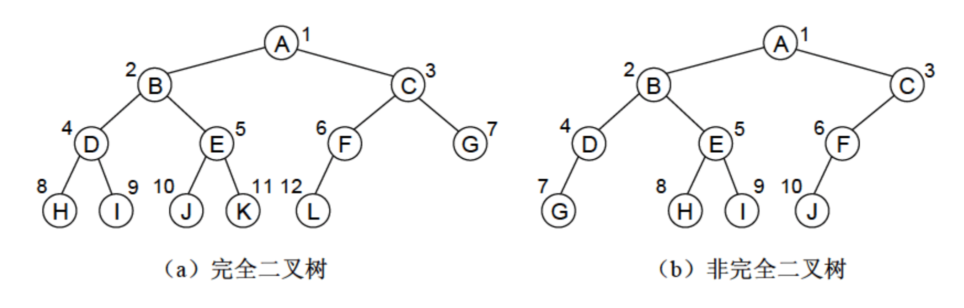


图 5.8 二叉树的层序编号

性质 5 在完全二叉树的层序编号中，对任一编号为 i 的结点 x（$1\leqslant i\leqslant n$），有：

（1）若 $i>1$，则 x 的双亲编号为$\lfloor i/2 \rfloor$；若 $i=1$，则 x 是根，无双亲。

（2）若 $2i>n$，则 x 无左孩子，否则其左孩子的编号为 2i。完全二叉树中的结点若无左孩子则肯定也无右孩子，即为叶子，故编号 $i>\lfloor n/2 \rfloor$的结点必定是叶子。

（3）若 $2i+1>n$，则 x 无右孩子，否则其右孩子的编号为 $2i+1$。

（4）若 i 为奇数且不为 1，则 x 的左兄弟编号为 $i-1$；否则无左兄弟。

（5）若 i 为偶数且小于 n，则 x 的右兄弟编号为 $i+1$；否则无右兄弟。

这一性质可用数学归纳法证明（略）。

性质 5 实际上指出了这样一个重要事实：完全二叉树中结点之间的父子关系可由它们层序编号间的关系来表达，如图 5.9 所示。

注意，该性质是对完全二叉树而言的，对非完全二叉树则不成立，如图 5.8（b）中，结点 D 的编号为 4，它的左孩子 G 的编号是 7 而不是 2×4=8。

如果结点编号从 0 开始，上述结果需要略作修改，如编号为 i（0≤i≤n−1）的结点，其双亲为⌊(i−1)/2⌋、左孩子为 2i+1 等。若无特别声明，本书中结点编号一般从 1 开始。

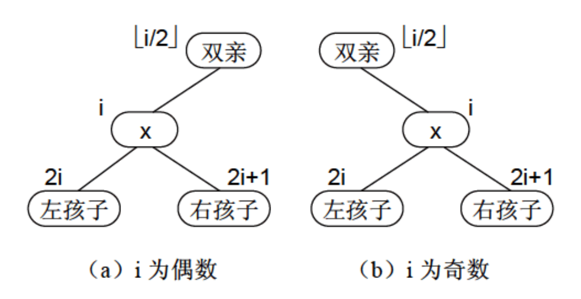


图 5.9　完全二叉树层序编号与父子关系

5.2.3　二叉树的存储

二叉树的存储结构应能体现二叉树的逻辑关系，即能反映结点的双亲和孩子关系。二叉树通常有两类存储结构：顺序存储结构和链式存储结构。

1．二叉树的顺序存储

二叉树的顺序存储就是将所有结点存储到一片连续的存储单元中，并能通过结点间的物理位置关系反映逻辑关系。这实际上就是要将二叉树的所有结点按一定次序排成一个线性序列，并且序列中结点间的次序关系要能反映结点间的逻辑关系。

由性质 5 知，完全二叉树结点的层序编号可反映结点间的逻辑关系，即由结点编号可推算出它的双亲和孩子的编号。所以，完全二叉树的结点可按层序编号的次序存储到一个向量 bt[1..maxsize]中，另外再指出结点数 n。由于 C/C++语言数组的下标从 0 开始，编号为 i 的结点将存放在下标为 i−1 的单元内，即结点编号和数组下标总要进行转换，不方便。为此，我们将数组定义为 bt[maxsize+1]，并从数组下标 1 开始使用。这就是完全二叉树的顺序存储结构，其 C/C++语言描述如下：

```
const int maxsize=100;      //结点数的最大值，假设为 100
typedef struct {
  datatype bt[maxsize+1];   //0 号单元未用
  int n;
} sqbitree;
```

其中数组 bt 的 0 号单元不用，但可用来存放其他信息，如结点数（这时数据域 n 可省略）。由于结点的编号就是数组的下标，所以通过下标间的数值关系就可知道结点之间的父子关系，即不需附加任何信息就可表示逻辑关系。例如，图 5.10 是图 5.8（a）完全二叉树的顺序存储结构示意图，其中 bt[5]的双亲是 bt[2]，其左、右孩子分别是 bt[10]和 bt[11]；bt[9]的双亲是 bt[4]，它没有孩子，因为 i=9，n=12，这时 2i>n。

显然，对完全二叉树而言，顺序存储结构既简单又节省存储空间。

然而，对一般的二叉树，结点间的层序编号关系并不能反映逻辑关系，所以不能直接

采用上述方法。但是，如果在二叉树上补充一些“虚结点”使其成为完全二叉树，则可使用上述方法了。由于要存储“虚结点”，会有一定的空间浪费，在最坏的情况下，一个深度为 k 且只有 k 个结点的右单支树却需要 2^k-1 个结点的存储空间。例如，只有 3 个结点 A、B 和 C 的右单支树，将其添上一些实际上并不存在的“虚结点”后，使它成为如图 5.11（a）所示的完全二叉树，相应的顺序存储结构见图 5.11（b）。图中虚线和“∧”表示虚结点。

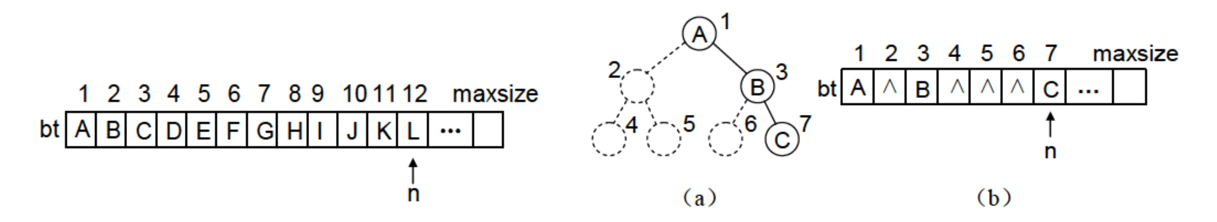


图 5.10 图 5.8（a）完全二叉树的顺序存储　　图 5.11 非完全二叉树的顺序存储

2. 二叉树的链式存储

从上面看到：用顺序方式存储一般的二叉树将浪费存储空间；另外，这种存储方式下在树中插入和删除结点也不方便。因此，树的最自然的存储方法是采用链式存储。二叉树链式存储的基本思想是，每个结点除了存放本身的数据外，还要根据需要设置指向双亲和左、右孩子的指针，即通过指针来反映逻辑关系。这样，根据指针的设置情况，存储方式可分为一指针式、二指针式和三指针式。具体选用何种方式，主要考虑运算实现的方便程度以及运算的频度。

与普通链式存储一样，二叉树的链式存储可以是“静态”的，也可以是“动态”的。常用的是动态链式存储，这时一般不用担心空间溢出问题，也不必关心存储管理的细节（由系统完成），这使我们能用更多的精力去考虑其他问题。在本章内我们以动态链表为主，也在适当地方介绍一下静态方法。

二叉链表是二叉树最常用的存储结构，其中每个结点除了存储结点本身的数据外，还设置两个指针域 lchild 和 rchild，分别指向该结点的左孩子和右孩子。这是二叉树链式存储的二指针形式。就像单链表由头指针唯一确定一样，一个二叉链表也由指向根结点的根指针唯一确定。为了某些运算的方便，也可给二叉链表增加头结点（但一般并没有这样做）。二叉链表的结点结构为：

lchild	data	rchild

二叉链表的类型定义如下：

```
typedef struct node * pointer;
struct node {
  datatype data;
  pointer lchild,rchild;
};
typedef pointer bitree;
```

与单链表类似，这里定义了两个相同的类型 pointer 和 bitree，前者用于指向链表中的一般结点，后者用于指向链表的根结点，即代表二叉链表。

图 5.12（b）就是图 5.12（a）所示二叉树的二叉链表。若二叉树为空，则 root=NULL。若结点的某个孩子不存在，则相应的指针为空。

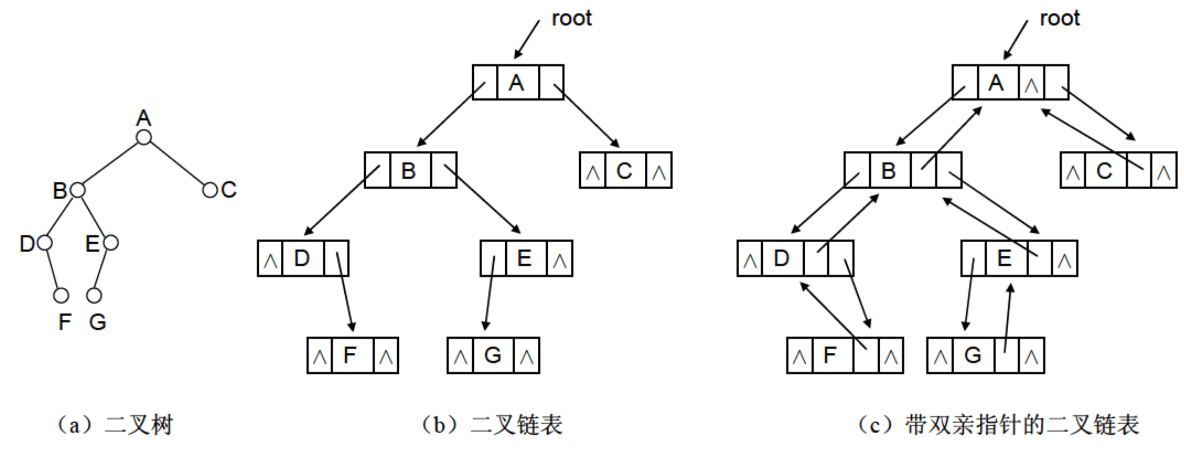


图 5.12　二叉链表

下面几节给出的有关二叉树的各种算法，大多数是基于这种存储结构的。如果经常要在二叉树中寻找某结点的双亲（或者祖先等），则可采用三指针形式：在每个结点上再增加一个指向其双亲的指针域 parent，形成一个带双亲指针的二叉链表[①]。图 5.12（c）就是图 5.12（a）所示二叉树的带双亲指针的二叉链表。至于一指针形式，就是在每个结点中只存放一个指向双亲的指针，这就是后面将要介绍的树的双亲表示法。

5.3　二叉树的遍历

在二叉树的一些运算中，经常要查找某种特征的结点，或对所有结点逐一进行某种处理，这就涉及二叉树的遍历问题，它是二叉树的一种重要运算。二叉树的**遍历**（Traversal）是指沿某条搜索路径周游二叉树，对每个结点访问一次且仅访问一次。这里的访问是指对结点进行某种处理，处理的内容依具体问题而定，可以是读、写、修改等。

我们知道，遍历一个线性结构很容易，只需从开始结点出发顺序扫描每个结点即可。但二叉树是一个非线性结构，每个结点可以有两个后继，因此，需要寻找某种规律来系统地访问树的各个结点。

5.3.1　二叉树的遍历方法

1. 递归遍历

根据定义，一棵非空的二叉树是由根结点、左子树、右子树这三个部分组成的，因此，遍历一棵非空二叉树的问题可分解为 3 个子问题：访问根结点；遍历左子树；遍历右子树。若分别用 D、L 和 R 表示上述 3 个子问题，则有 DLR、LDR、LRD、DRL、RDL、RLD

① 有的文献称之为“三叉链表”。

等 6 种次序的遍历方案。其中前 3 种方案是按先左后右的次序遍历根的两棵子树，后 3 种方案则是按先右后左的次序遍历根的两棵子树，由于二者对称，故我们只讨论前 3 种次序的遍历方案。

对遍历方案 DLR，访问根的操作是在遍历其左、右子树之前进行的，故称之为前序遍历（或先根遍历）。类似地，LDR 和 LRD 分别称为中序遍历（或中根遍历）和后序遍历（或后根遍历）。由于左、右子树本身也是二叉树，对它们的遍历也要按同样的规则进行处理，于是，二叉树的遍历就成为一个递归问题，因而很容易写出 3 种遍历方法的递归定义。

（1）前序遍历

若二叉树非空，则依次进行如下操作：

① 访问根结点。

② 前序遍历左子树。

③ 前序遍历右子树。

（2）中序遍历

若二叉树非空，则依次进行下列操作：

① 中序遍历左子树。

② 访问根结点。

③ 中序遍历右子树。

（3）后序遍历

若二叉树非空，则依次进行以下操作：

① 后序遍历左子树。

② 后序遍历右子树。

③ 访问根结点。

按以上 3 种方法遍历，都可得到所有结点的一个访问序列，分别称为先根（遍历）序列、中根（遍历）序列和后根（遍历）序列。例如，对图 5.12（a）所示的二叉树进行遍历，得到的先根序列为{A, B, D, F, E, G, C}；中根序列为{D, F, B, G, E, A, C}；后根序列为{F, D, G, E, B, C, A}。

由于上述遍历是递归定义的，故很容易写出相应的递归算法。显然递归终止条件是二叉树为空，此时应为空操作。假设对根结点的访问是打印结点数据，则在二叉链表上实现的 3 个遍历算法如下：

```
void preorder(bitree t) {   //先根遍历
  if(t==NULL) return;
  cout<<t->data<<endl;      //访问根
  preorder(t->lchild);      //先根遍历左子树
  preorder(t->rchild);      //先根遍历右子树
}
void inorder(bitree t) {    //中根遍历
  if(t==NULL) return;
  inorder(t->lchild);       //中根遍历左子树
  cout<<t->data<<endl;      //访问根
  inorder(t->rchild);       //中根遍历右子树
}
void postorder(bitree t) { //后根遍历
```

```
    if(t==NULL) return;
    postorder(t->lchild);      //后根遍历左子树
    postorder(t->rchild);      //后根遍历右子树
    cout<<t->data<<endl;       //访问根
}
```

为了便于理解上述 3 种递归算法，以先根遍历为例，对图 5.13（a）所示的二叉树，其先根遍历执行过程如图 5.13（b）所示，将其全部的递归调用关系画出来，得到图 5.13（c）。其中为简洁起见，将函数名 preorder 简写为 pre，将 lchild 和 rchild 分别简写为 l 和 r。虚线上的箭头表示各个递归调用的次序。可见，所有的调用过程（虚线）组成了遍历的搜索路线。递归调用关系完全可以画在原树上，得到更加简洁直观的图 5.14，其中“∧”表示虚结点。可以发现，搜索路线的特点是，从根结点出发，逆时针沿着二叉树外缘移动，每个结点均经过了 3 次。

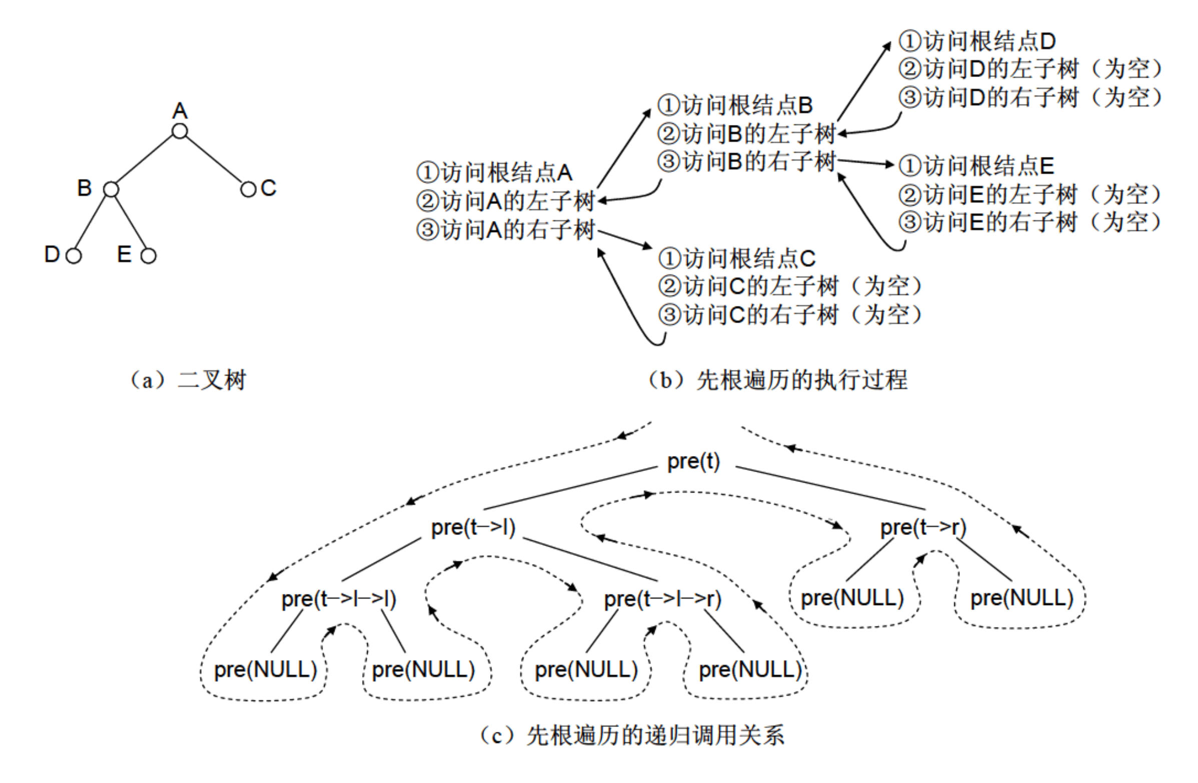


图 5.13 二叉树先根遍历的递归调用关系

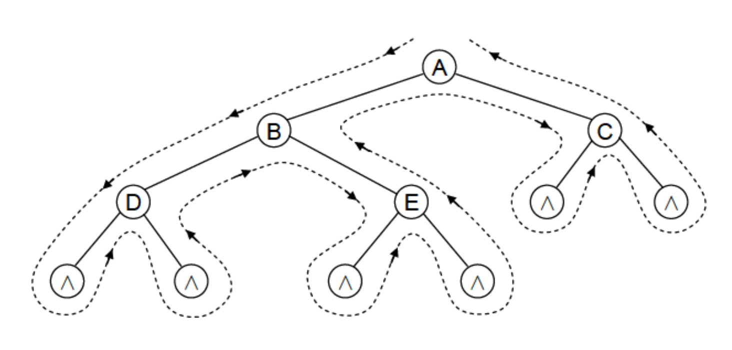


图 5.14 遍历二叉树的搜索路线

也可画出中根遍历和后根遍历的调用关系图，结果发现 3 种遍历的搜索路线是相同的。

这很好理解，因为，假设去掉对根的访问（或用空语句代替），3 种遍历算法就是一样的了，从而具有相同的搜索路线。

它们的区别在于访问结点的“时机”不同：若访问结点均是在第一次经过时进行，则是前序遍历；若访问结点均是在第二次（或第三次）经过时进行，则是中序遍历（或后序遍历）。因此，只要将搜索路线上所有在第一次、第二次和第三次经过的结点分别列表，即可分别得到该二叉树的前序序列、中序序列和后序序列。

遍历算法的基本操作是访问结点，显然，不论按哪种次序遍历，对含 n 个结点的二叉树，其时间复杂度均为 O(n)。所需辅助空间为递归遍历过程中栈的最大容量，即树的深度，最坏情况下为 n，则空间复杂度也为 O(n)。

中序序列具有下列特点（以后有关章节中会用到）：

① 中序序列的第一个结点是二叉树中最左下的结点。

② 中序序列的最后一个结点是二叉树中最右下的结点。

所谓**最左下**的结点，是指从根开始，沿左指针链往下查找，直到找到一个左指针为空（没有左孩子）的结点为止，这个结点就是“最左下”的结点。注意，最左下的结点虽然没有左孩子，但可能有右孩子（也可能无右孩子）。若无右孩子，则它必定是叶子。类似地，**最右下**的结点就是从根开始，沿右指针链往下查找，直到找到一个没有右孩子的结点为止，该结点是“最右下”的结点。

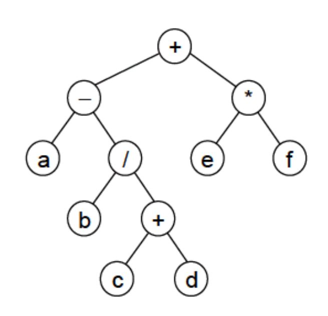
例 5.1 写出图 5.15 所示二叉树的先根、中根和后根遍历序列。

解：先根遍历序列为：+-a/b+cd*ef。

中根遍历序列为：a-b/c+d+e*f。

后根遍历序列为：abcd+/-ef*+。

实际上这种类型的二叉树称为**表达式树**，它的叶子结点表示操作数，分支结点表示操作符。如果一个表达式中的操作符都是双目或单目的，则一个表达式可对应到一棵二叉树。图 5.15 表示的实际上就是表达式 a−b/(c+d)+e*f。其先根、中根和后根遍历序列分别称为表达式的**前缀表示**（prefix，波兰式）、**中缀表示**（infix）和**后缀表示**（postfix，逆波兰式）：各操作符分别在对应操作数之前、之中和之后（但操作数的顺序相同）。



表达式 a−b/(c+d)+e*f 的二叉树

图 5.15 表达式树

中缀表示与原表达式一致，但无括号，这可对中根遍历略作修改：在遍历左子树前打印左括号，在遍历右子树后打印右括号，结果即得原表达式：(a−(b/(c+d))+(e*f))。后缀表示和前缀表示没有歧义，不必采用括号或优先级。

这里顺便说说表达式的求值。后缀表达式和前缀表达式的求值较简单：借助操作数栈，从左向右或从右向左扫描表达式，若遇到操作数，就入栈；若遇到操作符，则操作数出栈，由操作符运算后把结果作为操作数再入栈……。而中缀表达式的求值较麻烦，需要操作数和操作符两个栈，还要考虑各操作符的优先级（对扫描到的操作符分情况处理）。也可先转化为后缀表示或前缀表示后再求值，这要借助操作符栈，也要考虑各操作符的优先级。具体就不细述了。

2. 层序遍历

由于树形结构的结点间形成分支和层次关系，所以如果逐层地对结点进行访问，也可

将所有结点都访问到，这就是层序遍历。所谓二叉树的**层序遍历**，是指从第一层（即根）开始，按从上层到下层，每层内按从左到右的顺序对结点逐个访问。如对图 5.12(a)所示的二叉树，其层序遍历序列为 A, B, C, D, E, F, G。实际上，前面对二叉树结点进行层序编号的过程就是一种层序遍历，只是访问结点时的操作是给它一个编号而已。

由于上下层结点之间具有父子关系，在层序遍历中必然是先访问的结点其孩子结点也先访问，即若结点 A 先于结点 B 访问，则结点 A 的孩子也先于结点 B 的孩子访问。这样，在层序遍历中，我们可用一个队列来保存待访问的结点(已访问结点的孩子)。遍历算法为：每访问一个结点，就将它的孩子指针入队，下一个要访问的结点是队头（出队）；这个过程不断进行，直到队列为空。

除了第一个结点（即根）外，其他结点都是从队列中取出并进行处理的。为使根和其他结点的处理一致，可采用**预入队**技术，即在算法开始时先将根结点入队，然后马上出队再进行有关处理。非形式算法如下：

```
入队(根指针);
while(队不空) {
  出队(指针 p);
  访问 p;
  入队(左孩子);
  入队(右孩子);
}
```

假设对结点的访问是打印结点数据，并注意到空指针不必入队，则具体算法如下：

```
void levelorder(bitree t) {     //层序遍历
  pointer p;
  sqqueue Q;                    //循环队列，队列元素为结点指针，类型为 pointer
  if(t==NULL) return;
  init_sqqueue(&Q);
  en_sqqueue(&Q,t);             //根结点入队
  while(!empty_sqqueue(&Q)) {   //队列非空时
    de_sqqueue(&Q,&p);cout<<p->data<<endl;         //出队，访问队头
    if(p->lchild!=NULL) en_sqqueue(&Q,p->lchild);  //左孩子入队
    if(p->rchild!=NULL) en_sqqueue(&Q,p->rchild);  //右孩子入队
  }
}
```

对该算法进行修改，还可输出各结点的层数、树的高度、宽度等。

上面介绍的几种遍历序列都是线性序列，有且仅有一个开始结点和一个终端结点，其余结点都有且仅有一个前趋和一个后继。为了区别树形结构中前趋（即双亲）和后继（即孩子）的概念，对上述各种线性序列，我们在结点的前趋和后继之前冠以相应遍历次序的名称。例如，图 5.12（a）所示二叉树中，结点 B 的层序前趋是 A，层序后继是 C；中序前趋是 F，中序后继是 G 等。但就该树的逻辑结构而言，结点 B 的前趋是 A，后继是 D 和 E。

上述各种遍历都是基于二叉链表的，但显然也可在顺序存储结构上实现。这时，层序遍历由存储结构直接得到（去掉虚结点），递归遍历则注意利用顺序存储规律即可，如某树为空即该树根结点对应的单元为虚结点、结点 i 的左右孩子对应编号为 2i 和 2i+1 的单元等，具体算法就不细述了。

回顾一下链表，为了对所有结点进行某种处理，如累计结点数（求表长），需要从头

开始沿着指针链逐个结点进行，这实际上就是链表的遍历。一般而言，所谓遍历就是对所有数据结点按照某种方式无重复无遗漏地访问，即访问且只访问一次。数据结构中的很多算法，都是基于遍历的，大家在学习时要注意体会和掌握。

对非线性结构树以及下一章将要介绍的图，遍历后可得到所有结点按某种次序排列的一个线性序列，所以遍历也可看成是从非线性结构到线性结构的一种映射方法。

5.3.2 二叉树遍历与递归举例

二叉树的定义是递归的，所以二叉树的很多问题都可以很自然地进行递归处理，其中就包括遍历。本来，遍历只是这些递归处理中的一种，但如果我们广义地看待对根的访问，即相应的处理可以是任何操作，如输出其内容、某种性质的判断、某种信息的处理等，则很多问题都可以归结到遍历上。这样对同一个问题，我们可以直接用递归来处理，也可以用某种特殊的遍历来处理。一般来说，如果问题具有比较明显的逐个结点处理的特点，则用遍历比较直观；否则就按根、左子树、右子树三部分递归处理比较简便。

递归程序一般包括递归出口和递归体两部分。在二叉树中，递归出口一般是二叉树为空或叶子，此时不能再递归，只能回退；递归体一般是对根和左子树、右子树的某种处理。

例 5.2 求二叉树的叶子结点数。

解：这个问题具有明显的逐个结点处理的特点：如果当前结点是叶子，则叶子数加 1。所以可用各种遍历法，以先根遍历为例，算法如下：

```
int num=0;                 //num 为叶子数，设为全局量，并初始化为 0
void leaf(bitree t) {
  if(t==NULL) return;      //空树什么都不做
  if(t->lchild==NULL && t->rchild==NULL) num++;//访问根：若为叶子，则叶子数+1
  leaf(t->lchild);         //访问左子树：累计其中的叶子数
  leaf(t->rchild);         //访问右子树：累计其中的叶子数
}
```

上面将叶子数设成全局量而不作为参数，是考虑到递归函数执行时，参数要频繁传递，如果减少参数的个数，可提高递归函数的执行效率。

求叶子数问题，也可直接从递归的角度来考虑：二叉树的叶子数等于左子树和右子树的叶子数之和，但左子树和右子树本身又是二叉树，求其叶子数要递归进行。算法如下：

```
int leaf2(bitree t) {      //叶子数通过函数值返回
  int L,R;
  if(t==NULL) return 0;//空树的叶子为 0
  if(t->lchild==NULL && t->rchild==NULL) return 1; //树中只有一个点，就是叶子
  L=leaf(t->lchild);       //求左子树的叶子数(访问左子树)
  R=leaf(t->rchild);       //求右子树的叶子数(访问右子树)
  return L+R;              //叶子数=左右子树叶子数之和(访问根)
}
```

注意，该算法需要两个递归出口：空和叶子。另外，即使是这个算法，也可看成一种（后根）遍历：先访问子树（求子树叶子数），再访问根（将左、右子树的叶子数加起来作为当前子树的叶子数），但这显然不如直接按递归考虑容易理解。

按类似思想，可求二叉树的高度、度 1 结点数、度 2 结点数等。比如二叉树的高度是左、右子树中高度较大者加 1，但左、右子树本身又是二叉树，故递归。具体算法略。

例 5.3　交换二叉树各结点的左右子树。

解：这个问题显然可按遍历思想处理。以先根遍历为例，算法如下：

```
void exchange(bitree t) {   //先根遍历型
  pointer p;
  if(t==NULL) return;        //空树
  p=t->lchild;t->lchild=t->rchild;t->rchild=p; //交换
  exchange(t->rchild);       //遍历原左子树(现为 t->rchild)
  exchange(t->lchild);       //遍历原右子树(现为 t->lchild)
}
```

注意交换后左右顺序与原来相反，如访问原右子树时参数为 t->lchild。类似，若采用中根遍历，则在访问右子树时参数也应为 t->lchild（这时遍历左、右子树时的参数名称上都为 t->lchild，但含义不同，一个是交换前的，一个是交换后的），否则最终只交换了根结点的左右孩子。显然，采用后根遍历则不存在这些问题。

例 5.4　已知二叉树各结点存放的是字符，判断是否所有结点存放的都是数字字符。

解：这个问题显然可按逐结点遍历处理，但这里不一定要遍历完所有结点：一旦发现某个结点或子树不合要求，就不必对后继子树进行遍历了。算法如下：

```
int detect(bitree t) {
  int x;
  if(t==NULL) return 1;      //空树
  if(t->data<'0'|| t->data>'9')return 0;//访问根：若非数字字符，则跳过子树检查
  x=detct(t->lchild); if(x==0) return 0;//遍历左子树，若为假，则跳过右子树检查
  x=detct(t->rchild);        //遍历右子树
  return x;                  //最后结果由右子树决定
}
```

注意，该算法将空树看做真，否则还需增加一个递归到叶子（并判断真假）的递归出口。

本例也可直接用递归的思想处理：如果根不是数字字符，则当前结果肯定为假，返回；否则当前结果由左子树和右子树的情况共同决定。算法如下：

```
int detect2(bitree t) {
  if(t==NULL) return 1;                    //认为空树符合条件(真)
  if(t->data<'0'||t->data>'9') return 0;    //访问根：若不是数字字符则返回假
  return detct(t->lchild) && detct(t->rchild); //左子树和右子树共同决定
}
```

由逻辑与运算的短路规则，该算法的返回语句当检测到左子树为假时，并不会再检测右子树而直接返回假（实际执行过程与上一算法相同）。

例 5.5　判断两棵二叉树是否等价，即要么都为空；要么根相同，且左右子树分别等价。

解：这个问题比较适合直接按递归思想处理，算法如下：

```
int same(bitree t1,bitree t2) {
  if(t1==NULL && t2==NULL) return 1;    //同时为空树
  if(t1==NULL || t2==NULL) return 0;    //一个为空，另一个非空
```

```
    if(t1->data!=t2->data) return 0;      //根不相等
    return same(t1->lchild,t2->lchild) && same(t1->rchild,t2->rchild);
                                           //左子树和右子树共同决定
  }
```

注意，算法在检查两个根中一个空，另一个非空的条件不是(t1==NULL && t2!=NULL) || (t1!=NULL && t2==NULL)，而是 t1==NULL || t2==NULL，这是因为经过前一步两者同时为空的检查后，t1 和 t2 已不可能同时为空了，所以若某一个为空，则另一个必不为空。

另外，算法也利用了逻辑与运算的短路规则，如果左子树不等价，并不会检查右子树。

顺便指出，例 5.4 和例 5.5 对根和左、右子树的检查过程也可总体上写成一句的形式：return 根&&左&&右，但多个条件连在一起后反而显得不够简洁。

5.4 二叉树的生成

二叉树的生成是指如何在内存中建立二叉树的存储结构。建立顺序存储结构较简单，这里仅讨论如何建立二叉链表。要建立二叉链表，需要按某种方式输入二叉树的结点及其逻辑信息。注意到二叉树遍历时，不仅得到了结点信息，而且由序列中结点的先后关系还可获得一定的逻辑信息，如果这些信息足够，就可根据遍历序列生成相应的二叉树。

本节二叉树的生成方法就是基于遍历序列的，相当于遍历问题的逆问题，即由遍历序列反求二叉树，这需要分析和利用二叉树遍历序列的特点。以下分 3 种情况来讨论。

1．层序遍历序列

按完全二叉树的层次顺序，依次输入结点信息来建立二叉链表。这是因为完全二叉树的层序遍历序列中，结点间的序号关系可反映父子关系即逻辑关系。对一般的二叉树，要补充若干个虚结点使其成为完全二叉树后，再按其层次顺序输入。例如，仅含 3 个结点 A、B、C 的右单支树（见图 5.11），按完全二叉树的形式输入的结点序列为：A@B@@@C#，其中@表示虚结点，#表示输入结束。

算法的基本思想是：依次输入结点信息，若输入的结点不是虚结点，则建立一个新结点；若新结点是第 1 个结点，则令其为根结点；否则将新结点作为孩子链接到它的双亲结点上。如此重复下去，直至输入字符“#”为止。

这里的关键是新结点与其双亲的链接。由于结点是按层次自左向右输入的，所以先输入的结点，其孩子也必定较先输入。即结点与其孩子具有先进先出的特点，于是可设置一个队列，保存已输入结点的地址。这样，队尾是当前正输入的结点，队头是其双亲结点。当队头结点的两个孩子都输入完毕后，出队，新的队头是下一个要输入孩子的双亲结点。如此下去，直到输入结束符为止。

双亲与孩子的链接方法是：若当前输入的结点编号是偶数，则该结点作为左孩子与其双亲链接；否则作为右孩子与其双亲链。若双亲结点或孩子结点为虚结点，则无需链接。

如果采用顺序队列，则队列是一个指针数组。为使队列元素在数组中的下标与其结点的层序编号一致，数组从下标 1 开始使用。注意，这里不是循环队列。具体算法如下：

```
bitree level_creat() {      //按层序序列建立二叉树，返回根指针
  char ch;
```

```
  pointer Q[maxsize+1];      //非循环队列，有效下标从 1 到 n，maxsize 为最大结点数
  int front,rear;
  pointer root,s;
  root=NULL;                 //置空二叉树
  front=rear=0;              //置空队列
  while(cin>>ch,ch!='#') {   //输入字符，若不是结束符则循环
    if(ch!='@') {            //非虚结点，建立新结点
      s=new node;
      s->data=ch;
      s->lchild=s->rchild=NULL;
    }
    else s=NULL;
    rear++;Q[rear]=s;                  //不管结点是否为虚，都要入队
    if(rear==1) {root=s;front=1;}      //第一个结点是根，要修改头指针，它不是孩子
    else {
      if(s && Q[front])                //孩子和双亲都不是虚结点，链接之
        if(rear%2==0) Q[front]->lchild=s;    //rear 是偶数，新结点是左孩子
        else          Q[front]->rchild=s;    //rear 是奇数，新结点是右孩子
      if(rear%2==1) front++;           //不论是否为虚，右孩子入队后，双亲出队
    }
  }
  return root;
}
```

2．先根、中根或后根遍历序列

基本思想是输入这 3 个遍历序列中的一个，二叉树的结点就按相应的遍历过程逐个生成。类似于层序遍历，如果不对遍历序列作些补充，是不能完整地反映结点间的逻辑关系的，也就不能得到正确的结果。补充的方法也是增加虚结点，但这里只需对空指针对应的位置进行补充，而不必补充到完全二叉树的形式。

以先根遍历和图 5.12(a)为例，二叉树的先根输入序列为 ABD@F@@EG@@@C@@，其中@表示虚结点。注意，这里不需要结束符。算法过程为，先生成根结点，再生成左子树，然后是右子树，左右子树的生成采用递归，算法如下：

```
bitree pre_creat() {          //由先根序列建立二叉树，返回根指针
  bitree t;
  char ch;
  cin>>ch;
  if(ch=='@') return NULL;    //虚结点
  t=new node;                 //生成根结点
  t->data=ch;
  t->lchild=pre_creat();      //生成左子树
  t->rchild=pre_creat();      //生成右子树
  return t;
}
```

该算法的调用形式为“bitree T; T=pre_creat();”等。

若给定的是后根序列，由于第一个结点就是虚结点，为递归出口，于是后面递归体一次也不会执行，从而生成不了二叉树。但把序列倒过来看，则相当于按“根→右→左”的顺序遍历原二叉树，于是可把原序列逆置后，按“根→右→左”的“先根遍历”来生成二叉树。具体算法略。但是，若给定的是中根序列，除了第一个结点为虚结点的问题外，更

主要是因为不能确定谁是根，二叉树不唯一，从而不能生成二叉树。

3. 双遍历序列

上面看到，单个的遍历序列一般需要补充虚结点才能完整表示结点间的逻辑关系。但两个或多个信息不完整的序列如果能相互补充，也可能得到完整的信息。这需要分析和利用遍历序列的特点。由遍历方法得到以下几点是显然的：

（1）对前序序列，序列的第一个结点就是整个二叉树的根。

（2）对后序序列，序列的最后一个结点就是整个二叉树的根。

（3）对中序序列，以根为界，序列的前一部分为根的左子树，后一部分为根的右子树；并且，由前一部分构成的子序列是左子树的中序序列，由后一部分构成的子序列是右子树的中序序列。

若给定了前序序列和中序序列，反复利用上面的（1）和（3），就可获得完整的逻辑信息，即由前序序列找到根，由中序序列得到左、右子树；再对每个子树由前序序列找到子树的根，由中序序列得到子树的左、右子树，……，依此类推，每次得到一个结点（子树的根），从而逐渐分离出树的全部信息，也就可以还原和构造出该二叉树。这个过程参见图 5.16。

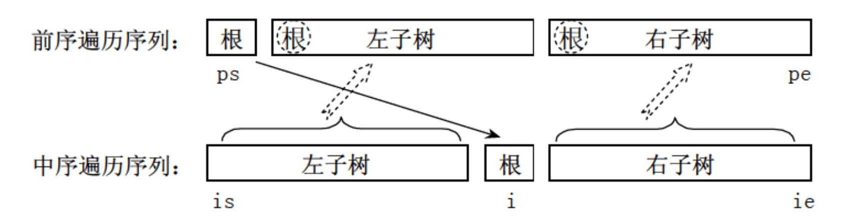


图 5.16　由前序和中序序列建立二叉树

由于构造过程是递归的，可方便地写出相应的递归算法：

```
bitree creat(char Pre[],int ps,int pe,char In[],int is,int ie) {
//由先序和中序建立二叉树，Pre[]和 In[]为遍历序列，ps,pe 以及 is,ie 分别为区间始末点
  bitree t;
  int i;
  if(ps>pe) return NULL;
  t=new node;                  //生成根结点
  t->data=Pre[ps];
  i=is;
  while(In[i]!=Pre[ps]) i++;    //寻找中序序列中根的位置 i
  t->lchild=creat(Pre,ps+1,ps+i-is,In,is,i-1); //生成左子树
  t->rchild=creat(Pre,ps+i-is+1,pe,In,i+1,ie); //生成右子树
  return t;
}
```

注意，在递归中要用参数指明当前正在处理的子树的中序与前序序列在原来整个中序与前序序列中的位置。

一般地，由中序序列与前序序列、中序序列与后序序列、中序序列与层序序列等都能唯一确定二叉树。而由前序序列和后序序列因不能确定左右子树，一般就不能唯一确定二叉树（除非二叉树为空或只有一个结点；或补充其他条件如为严格二叉树等）。比如由前序序列{A, B, D, C}和后序序列{D, B, C, A}可得到如图 5.17 所示的两棵二叉树。

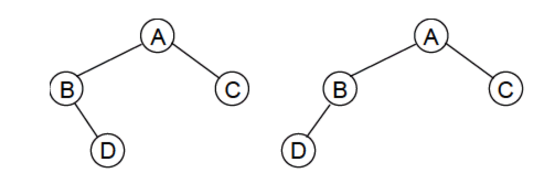


图 5.17　前序序列和后序序列分别相同两棵二叉树

如果要把一棵二叉树存储到外存（文件）中，我们就可只存储它的中序序列与前序序列，或者中序序列与后序序列等，以后在需要时再将二叉树还原出来。这样，既不必像顺序存储那样，存储大量虚结点，又不必像二叉链表那样，存储大量附加指针。因此，也可看成是二叉树的一种存储方法。

5.5 递归消除

在 5.3.1 节中，我们由遍历的递归定义直接写出了遍历的递归算法。一般地，对一个递归问题，设计出求解该问题的递归算法是比较容易的，并且一般也有较好的可读性。然而，有时需要考虑怎样将一个递归算法转换成等价的非递归算法，这称为“**递归消除**（recursion removal）”。研究递归消除的原因主要有以下几个方面：

第一，递归消除有利于提高算法的时空性能。一般而言，递归算法的时空性能相对较差，非递归算法的时空性能要高得多。比如，递归程序执行时需反复调用自己，与调用其他函数一样，每调用一次，系统就在其运行栈中存放函数的返回地址、参数、临时变量等，随着递归的进行，运行栈不断增长，只有当函数彻底执行完后，才释放它所占用的栈空间。因此，递归算法不但不节省空间，而且一般还比非递归算法浪费空间。如果某个递归算法的时空性能较差，又要被频繁地使用，则将它转换为非递归算法的意义就更大了。

第二，研究递归消除有助于透彻理解递归机制，而这种理解是熟练掌握递归程序设计技能的必要前提。

第三，有些程序设计语言不允许递归，如 FORTRAN 等高级语言和大多数汇编语言，这时就不得不考虑非递归算法。

一般说来，如果将原来由系统管理的递归程序的工作过程（主要是工作栈的工作过程）改为由程序员来模拟和管理，都可实现递归消除，但这并不是说递归消除一定要利用栈；即使要用栈，也不一定要将工作栈的工作过程都模拟出来，比如，若保存的现场信息在返回后不再使用，则可不必保存，从而提高程序效率。本节根据是否需要引入工作栈作为控制机构，将递归消除技术分成两类加以讨论。

5.5.1 简单递归消除

对于简单的递归算法，可以不通过工作栈作控制机制而直接转换成循环算法。这时计算依赖图（函数调用关系图）的分析和化简是一种非常有用的辅助手段。下面通过两个例子来进行说明。

例 5.6　将计算阶乘 n!的递归算法转换成非递归算法。

解：求阶乘的递归算法为：

```
long f(int n) {
  if(n==0) return 1;
  else return n*f(n-1);
}
```

计算依赖图的画法是：对每一个递归调用，在图中用一个结点来表示；对任意两个递归调用 p、q，若 p 的计算直接依赖于 q，即 p 的执行中调用了 q，则在 p 和 q 之间画一条线，并且 p 点的位置高于 q 点。通常只能且只需画出计算依赖图的一部分。

对于计算 n!的递归算法 f(n)来说，以 n=3 为例，f(3)的计算直接依赖于 f(2)，因为 f(3)调用了 f(2)，仅当 f(2)的计算完成之后，f(3)的计算才能完成。类似地，f(2)直接依赖于 f(1)；f(1)直接依赖于 f(0)；而 f(0)不依赖于其他调用，计算由自己独立完成。包含这几个递归调用的计算依赖图如图 5.18 所示。图中虚线表示递归调用和返回的次序，向下的箭头表示递归调用；向上的箭头表示返回。根据计算依赖图的画法和含义，这些虚线和箭头可以省略。

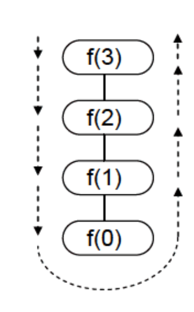


图 5.18 f(n)的部分计算依赖图

对照图 3.6（a）不难看出，图 5.18 只不过是它的一种简化形式。事实上，计算依赖图集中反映了不同递归调用之间的依赖关系。通过对这种关系的分析，可以很方便地写出完成同样任务的循环算法。

从图 5.18 中可以明显看出，递归算法 f(n)的计算实际上由一个自上而下的递归调用阶段和一个自下而上的返回阶段构成；并且所有递归调用都直接或间接地依赖于 f(0)。因此，整个计算完全可以只由后一个阶段完成，即先后依次计算 f(0)、f(1)、f(2)、…、f(n)。这是一个**递推**（recurrence）过程，可用循环完成。引入一个工作变量记录中间结果，则计算 n!的循环算法为：

```
long f1(int n) {
  long x;
  int i;
  x=1;
  for(i=1;i<=n;i++) x=x*i;
  return x;
}
```

显然，该算法的时间复杂度为 O(n)。

作为对比，这里分析一下前面的递归算法 f(n)。在求 f(n)时需先求 f(n−1)；若已知 f(n−1)，则其他运算（乘法和返回）的时间为常量，设为 c，即 T(n)=T(n−1)+c，但 n=0 时不递归直接返回结果，执行时间也为常量，设为 d，即 T(0)=d。于是：

$$
\begin{aligned}
T(n) = T(n-1)+c \quad &= [T(n-2)+c]+c \\
&= T(n-2)+2c = [T(n-3)+c]+2c \\
&= T(n-3)+3c = \cdots \\
&= T(0)+nc \\
&= d+nc = O(n)
\end{aligned}
$$

可见这里递归和非递归算法的时间复杂度相同，但显然非递归算法没有函数调用和返

回等时空开销，其实际执行效率要高些。

上述问题比较简单，计算依赖图中没有分支（**单向递归**），且递归调用语句是递归程序的最后一句（**尾递归**），熟练后这类问题可不画计算依赖图而直接用循环消除递归（得到迭代算法）。在较复杂的情况下，计算依赖图为递归消除提供了一种直观的分析工具。

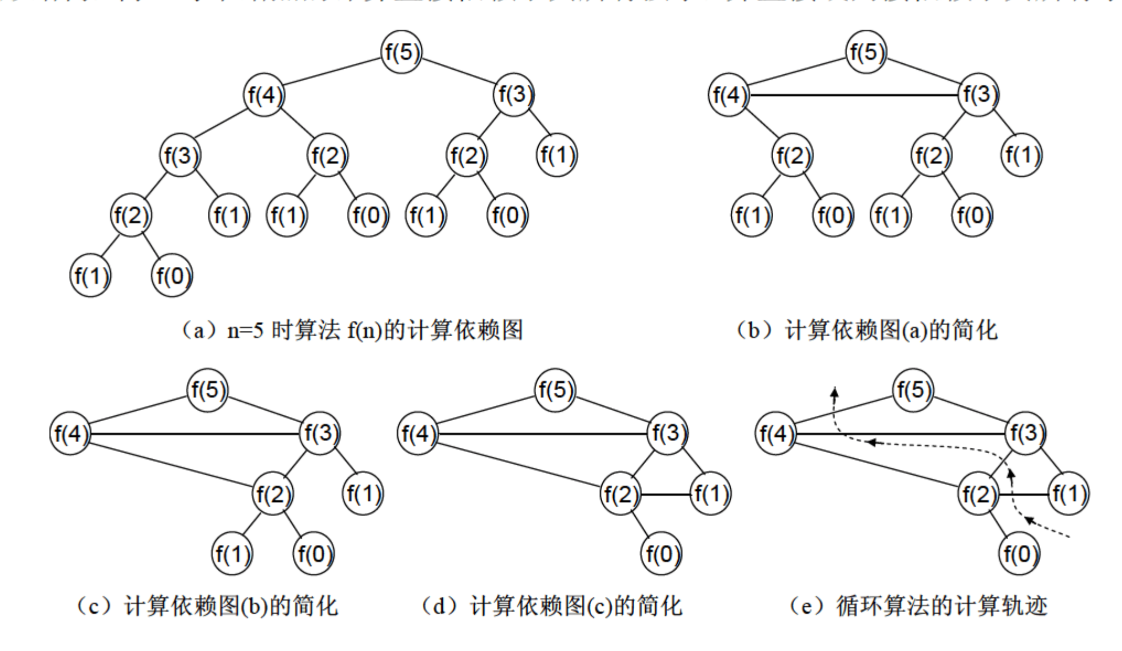
例 5.7　写出计算菲波那契数列的递归函数并消除递归：

$$\mathrm{Fib}(n)=\begin{cases}0 & n=0\\ 1 & n=1\\ \mathrm{Fib}(n-1)+\mathrm{Fib}(n-2) & n>1\end{cases}$$

解：由于 Fib(n)是递归定义的，可直接写出它的递归算法如下：

```
int f(int n) {
  if(n==0) return 0;
  else if(n==1) return 1;
  else return f(n-1)+f(n-2);
}
```

n=5 时算法 f(n)的计算依赖图见图 5.19（a）所示。显然，f(n)的计算依赖关系是一个树形结构，树上每个结点的计算直接依赖于其所有孩子，并直接或间接依赖于其所有子孙。



（a）n=5 时算法 f(n)的计算依赖图　（b）计算依赖图(a)的简化

（c）计算依赖图(b)的简化　（d）计算依赖图(c)的简化　（e）循环算法的计算轨迹

图 5.19　算法 f(n)的部分计算依赖图及其简化

从图 5.19（a）可以看出，递归算法 f(n)的效率不高。其中，f(3)被计算了两次，f(2)被计算 3 次，f(1)和 f(0)分别被计算 5 次和 3 次。可以证明（类似附录 E 中菲波那契数列通项的推导），计算 f(n)时所有递归调用的总次数为指数级 $O(2^n)$，效率是非常低的。

显然，如果能把重复的结点合并在一起，就可避免重复计算，从而提高效率。这可自上而下地按下列步骤得到：第一步，将两个 f(3)结点合并，得到子图（b）所示的计算依赖图；其次，将子图（b）中的两个 f(2)结点合并得子图（c）；最后，将子图（c）中的两个 f(1)结点合并，得到子图（d）。这个过程就是计算依赖图的化简。

由图 5.19（d）可知，当 n≤5 时，f(n)的全部计算只依赖于 f(0)和 f(1)。易知此结果对

任意 n≥2 都成立。因此，f(n)的计算可以通过自下而上的循环算法完成；相应的计算轨迹可设计成子图（e）中的虚线。由于当 n≥2 时，每个 f(n)直接依赖于刚刚计算出的前两个值，故引入两个工作量 x、y 分别记录中间结果。具体算法如下：

```
int f1(int n) {
  int x,y,z,i;
  if(n==0) return 0;
  if(n==1) return 1;
  x=0;y=1;
  for(i=2;i<=n;i++) {
    z=x+y;
    x=y;y=z;
  }
  return z;
}
```

在计算 f(n)的过程中，对每个 f(i)（i=0, 1, …, n），此算法只计算一次，故其时间性能明显高于递归算法 f(n)。

一般地，计算依赖图都可看成树型结构（图 5.18 为单支树），故也常称**递归树**（Recursion Tree），它是分析递归算法的一个重要工具，如树的高度对应递归深度，决定了递归所需栈空间的大小；树中结点数对应递归调用次数，决定了递归所需时间的多少等。

5.5.2 基于栈的递归消除

在很多情况下，递归算法的计算依赖图无法简化成线性结构（如图 5.18）或准线性结构（如图 5.19（d）），因而无法直接转换成循环算法。例如，从表面上看，二叉树的 3 种遍历的递归算法与计算 f(n)的递归算法似乎并无太大的区别，只是过程体中包含两处递归调用。然而，遍历算法的计算依赖图是不能简化的，比如在图 5.13（c）和图 5.14 中，各分支结点的实参实际上是不同的（对应二叉树上的不同结点），故不能合并化简。其中，图 5.14 正是先根遍历算法的计算依赖图。

在这种情况下，通常引入一个工作栈作为控制机构以消除递归。在例 3.3 中曾提到，编译系统利用工作栈来保存返回位置以实现过程调用与返回控制，这一思想同样适用于递归消除。下面以先根遍历（以下用 pre 表示）为例，具体讨论如何用工作栈来消除递归。

首先要弄清工作栈的作用方式，即工作栈怎样控制先根遍历的走向。图 5.20（a）所示为二叉链表上的任一结点 x 以及它的左、右子树 X_L 和 X_R。假设 t 是指向结点 x 的指针，则与子图（a）相应的有关递归调用如子图（b）所示。由于是先根遍历，当遍历到结点 x 时，即执行 pre(t)时，有 3 项工作需顺序完成：

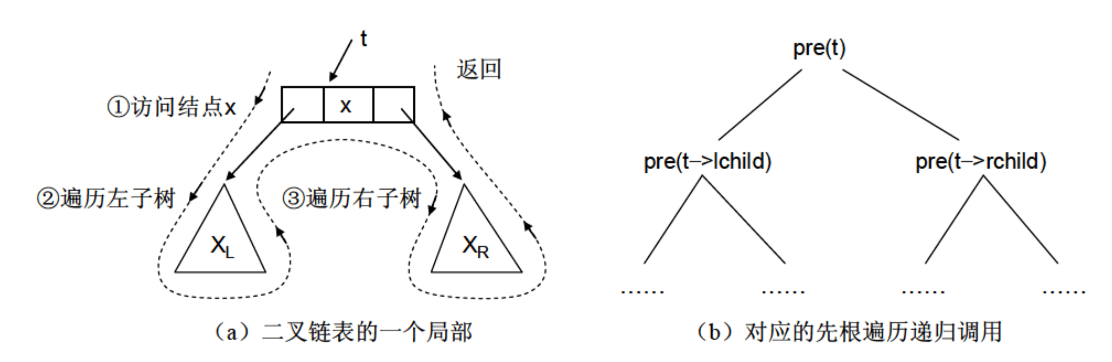
（1）访问结点 x（子树的根）。

（2）遍历 X_L，即调用 pre(t–>lchild)。

（3）遍历 X_R，即调用 pre(t–>rchild)。

其中，（1）和（2）的连接没有问题，但（2）与（3）如何连接需要考虑。为了执行（3），必须知道 X_R 的根指针，即结点 x 的右指针 t–>rchild；但 x 的左子树 X_L 上并没有这个指针。所以在执行（2）之前应将指针 t–>rchild 保存起来；在任务（2）完成之后再取出该指针以执行任务（3）。此后，指针 t–>rchild 就没有保存价值了。对于先根遍历来说，完成

了任务（3）也就完成了对以结点 x 为根的整个子树的遍历，接下来便是退回到 x 的父结点。



（a）二叉链表的一个局部　　（b）对应的先根遍历递归调用

图 5.20　先根遍历的分析

在整个遍历过程中，较后访问的结点在以后返回时较先返回，相应地，其右子树也较先访问。所以应以栈的方式保存结点信息。

以上保存的是 t–>rchild。也可保存结点本身的地址 t，以后再通过 t–>rchild 访问其右子树，但显然不及前者好。引入工作栈后，非递归算法在二叉链表的任一结点 x 上的主要操作步骤可归纳如下：

（1）若结点 x 不空，则访问结点 x：visit(p)。

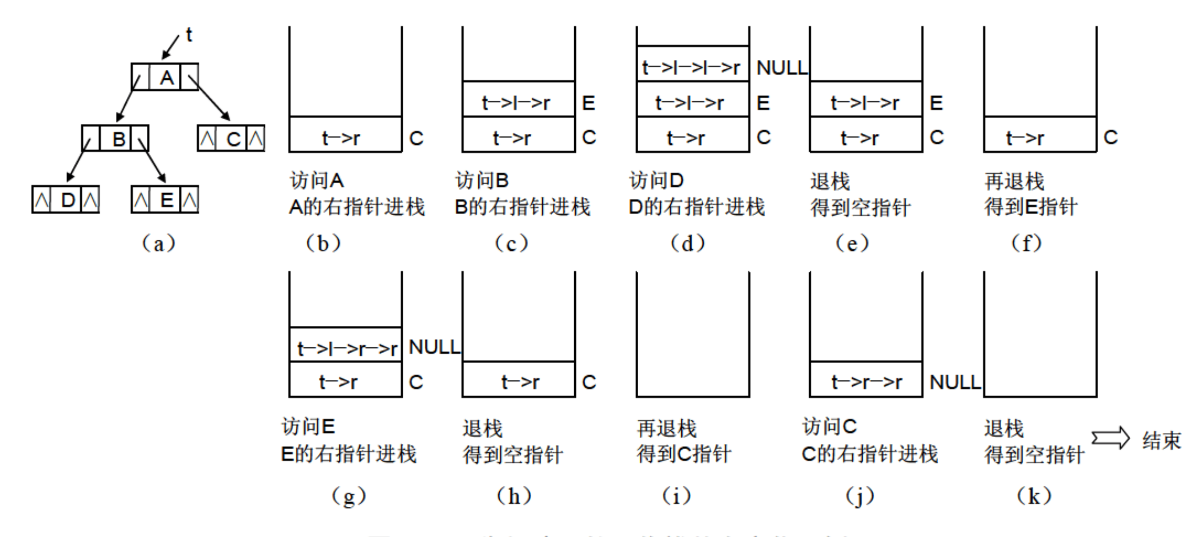
（2）结点 x 的右指针进栈：push(p–>rchild)。

（3）遍历 X_L：p=p–>lchild，转（1）。

（4）退栈，从栈中得到 x 的右指针：pop(p)。

（5）遍历 X_R：转（1）。

由于子树也是二叉树，所以步骤（3）、（5）对子树的操作即将流程控制转（1）。对图 5.21（a）所示的二叉链表，按上述步骤执行的主要过程如图 5.21（b）～图 5.21（k）所示。图中 lchild 和 rchild 分别缩写为 l 和 r。



（a）（b）（c）（d）（e）（f）（g）（h）（i）（j）（k）

图 5.21　先根遍历的工作栈状态变化示例

从中可见，在执行过程中可能多次出现栈空，但中间几次栈空时取出来的指针不空，只有最后栈空且取出来的指针也空时才终止算法。整个算法如下：

```
void preorder(bitree t) {   //先根遍历算法，非递归
  pointer p,S[maxsize];     //顺序栈
  int top;                  //栈顶指针
  if(t==NULL) return;
  top=-1;
  p=t;
  while(!(p==NULL && top==-1)) {
    while(p!=NULL) {
      visit(p);             //访问根
      S[++top]=p->rchild;   //右指针入栈
      p=p->lchild;          //向左搜索
    }
    p=S[top--];
  }
}
```

该算法的遍历过程如下：从根开始，顺左分支往下搜索，每搜索到一个结点，就访问该结点，同时将其右指针进栈，直到左分支为空。然后，退栈，取出最近保存的一个右指针，如果为空，再退栈取出另一个右指针；否则对右指针对应的右子树进行同样处理。如此一直进行下去，直到所有结点均已处理完。其中，退栈及其以后的操作，对应于递归遍历算法中，返回到已搜索过的最近一个（有右孩子的）结点，然后再对其右子树进行递归处理的过程。

该算法最先访问的是根，它的处理与其他结点是有所不同的：根的值是直接给定的，其他结点的值则是循环中从栈或左指针得到的。为使根和其他结点的处理方式一致，并简化循环条件，可采用**预入栈**技术，即在算法开始时先将根结点入栈，然后马上出栈进行有关处理，整个算法相当于从上述步骤（4）开始。非形式算法和求精后的算法如下：

```
push(根指针);
while(栈不空) {
  pop(指针 p);
  while(p!=NULL) {
    访问 p;
    push(p->rchild);
    p=p->lchild;
  }
}
void preorder2(bitree t) {  //先根遍历算法，非递归，根预入栈
  pointer p,S[maxsize];     //顺序栈
  int top;                  //栈顶指针
  if(t==NULL) return;
  top=-1;
  S[++top]=t;               //根指针入栈
  while(top>=0) {           //栈非空
    p=S[top--];             //出栈
    while(p!=NULL) {
      visit(p);             //访问根
      S[++top]=p->rchild;   //右指针入栈
      p=p->lchild;          //向左搜索
    }
  }
}
```

注意，对该算法，在图 5.21（b）之前还有一个根指针进栈和出栈的过程。

如果将算法中的赋值语句 p=p->lchild 通过栈来等效完成 S[++top]=p->lchild，p=S[top--]，并注意到空指针不必入栈，则上述算法的两个循环可合为一个，算法如下：

```
void preorder3(bitree t) { //先根遍历算法，非递归，根预入栈，改进
  pointer p,S[maxsize];     //顺序栈
  int top;                  //栈顶指针
  if(t==NULL) return;
  top=-1;
  S[++top]=t;               //根指针入栈
  while(top>=0) {           //栈非空
    p=S[top--];             //出栈
    visit(p);               //访问根
    if(p->rchild!=NULL) S[++top]=p->rchild;//右指针入栈
    if(p->lchild!=NULL) S[++top]=p->lchild;//左指针入栈
  }
}
```

中序遍历的非递归算法也很容易。中序遍历时，要先访问左子树，所以每搜索到一个结点时并不立即访问它，只有左子树访问完后下一步才访问根。于是，对搜索到的每一个结点，应当保存（入栈）当前结点自己的指针（而不是其右指针），以便搜索回退时得到需要的返回位置（出栈），再访问之。算法如下：

```
void inorder(bitree t) {    //中根遍历算法，非递归
  pointer p,S[maxsize];     //顺序栈
  int top;                  //栈顶指针
  if(t==NULL) return;
  top=-1;
  p=t;
  while(!(p==NULL && top==-1)) {
    while(p!=NULL) {        //搜索到最左下的结点
      S[++top]=p;
      p=p->lchild;
    }
    p=S[top--];
    visit(p);               //访问根
    p=p->rchild;            //向右搜索
  }
}
```

后序遍历的非递归算法要复杂一些。在后序遍历时，最后访问根结点，所以对任一结点，应先沿它的左分支往下搜索，每搜索到一个结点就进栈，直到左分枝为空；然后退栈，取出最后入栈的结点 x，但此时并不访问它，而是从该结点的右分支（若有的话）的根开始，按同样的方法沿它的右分枝处理；处理完结点 x 的右分枝后才能访问结点 x。

因此，任一结点进栈后，都有两次退回到栈顶：第一次是在处理完它的左分支时，第二次是在处理完它的右分枝时。显然，根结点只有在第二次退回到栈顶时才出栈并访问它。为了区分是第几次退回到栈顶，一般有两种方法。

（1）检查一下刚访问的结点是否是栈顶的右孩子，算法如下：

```
void _postorder(bitree t) {//后根遍历，非递归，无进栈标志
  pointer p,q,S[maxsize];   //顺序栈
  int top;
  if(t==NULL) return;
```

```
  top=-1;
  p=t;q=NULL;                //左子树尚未遍历，最近已访问结点 q 为空
  while(!(p==NULL && top==-1)) {
    while(p!=NULL) {
      S[++top]=p;
      p=p->lchild;           //向左搜索
    }
    p=S[top--];              //退栈
    if(q!=p->rchild) {
      S[++top]=p;            //再进栈
      p=p->rchild;           //转右子树
      q=NULL;                //右子树尚未遍历，最近已访问结点 q 为空
    }
    else {
      cout<<p->data<<" "<<endl; //访问根
      q=p;p=NULL;
    }
  }
}
```

其中，q 记录左子树或右子树最近已访问过的结点（子树还未访问时 q 为空）。另外，再进栈过程可省，改为先检测栈顶（取栈顶），而不是先退栈。这里是为了与下述采用标志 flag 的算法结构上一致。

（2）为每个结点设个进栈标志 flag，并随结点一起进出栈。非形式算法如下：

```
p=根;
while(结点未遍历完) {
  while(p 非空) {push(p,flag=1);p=p->lchild;}  //向左搜索
  pop(p,flag);                                 //退栈
  if(flag==1) {push(p,flag=2);p=p->rchild;}    //再进栈，转右子树
  else {visit(p);p=NULL;}                      //访问根
}
```

其中访问根后，置 p=NULL 是为了下一步继续退栈（回退）。这里入栈元素是一个结构体：(指针, flag)，也可单独设标志栈，并与结点栈同步进出。算法略（略显烦琐）。

通过前述例子可见，工作栈在消除递归中的基本作用是提供一种控制机制。在非递归算法执行过程中的某些关键时刻，用栈顶元素来“引导”下一步操作的“走向”。为此必须提前将有关信息进栈保存。例如上面先序遍历的例子，工作栈保存的是各个结点的右指针，也就是该结点右子树的根指针。显然，这些根指针正是先根遍历递归算法中包含的一些递归调用的实参；这些实参在非递归算法中若不及时保存就会丢失。因此，递归算法中的调用与返回控制被工作栈的作用所取代，从而将递归算法转换成非递归算法。

有以下几点需要注意：

（1）图 5.19（a）计算菲波那契数列的递归过程可看成二叉树的后根遍历，据此也可写出利用栈的非递归算法。

（2）由于中序遍历和后序遍历时最先访问的不是根，所以它们相应的非递归算法中没有采用根预入栈技术（也不合适）。

（3）以上通过栈进行递归消除后算法变得复杂了，而时空性能基本未变（相当于把系统栈改为人工管理，可能节省一些不必要的操作，但时间复杂度数量级不变，具体分析略），

似乎意义不大。这里主要是介绍递归消除的原理。当然，对不允许递归的语言就有意义了。

（4）如果采用带双亲指针的二叉链表，递归消除时可不用栈，因为可由双亲指针进行回退。对下面将介绍的线索二叉树，遍历时也可不需要栈。另外，即使需要递归栈，栈空间也可不单独分配，而在原结点空间上就地进行：在向左下结点搜索时，每搜索到一个结点，先将其左孩子指针取出，然后在该处存入其双亲指针，这样在搜索结束后就可由双亲指针进行回退。当然，在回退的同时要恢复原孩子指针。这些内容就不细述了。

5.6 线索二叉树

当用二叉链表作为二叉树的存储结构时，因为每个结点中只有指向其左、右孩子结点的指针域，所以从任一结点出发只能直接找到该结点的左、右孩子，一般情况下无法直接找到该结点在某种遍历序列中的前趋和后继结点。但是，若在每个结点中增加两个指针域来存放遍历序列的前趋和后继信息，又将大大降低存储空间的利用率。注意到 n 个结点的二叉链表中含有 n+1 个空指针域[①]，于是可利用这些空指针域来存放某种遍历次序下的前趋和后继结点的指针。这种附加的指针称为“线索”，加上线索的二叉链表称为线索链表，相应的二叉树称为**线索二叉树**（Threaded Binary Tree）。

这时，结点中的指针可能指的是孩子，也可能指的是线索，为了区分，可在结点中设置两个线索标志位，结点结构为：

lchild	ltag	data	rtag	rchild

其中：

$$\text{左线索标志ltag}=\begin{cases}0 & \text{lchild是指向结点的左孩子的指针}\\1 & \text{lchild是指向结点的前趋的左线索}\end{cases}$$

$$\text{右线索标志rtag}=\begin{cases}0 & \text{rchild是指向结点的右孩子的指针}\\1 & \text{rchild是指向结点的后继的右线索}\end{cases}$$

以图 5.22（a）所示的中序线索二叉树为例，它的线索链表见图 5.22（d），其中实线表示指针，虚线表示线索。其中结点 D 的左线索为空，表示它是中序序列的开始结点，没有前趋；结点 C 的右线索为空，表示它是中序序列的终端结点，没有后继。显然在线索二叉树中，一个结点为叶结点的充要条件为：它的左、右线索标志均是 1。

从图 5.22 中可发现如下特点：中序线索一般都是“向上”指的，即指向其祖先结点；而先序和后序线索就不一定（见图 5.22（b）和图 5.22（c）），可以向上指，也可以向下指，还可以同级指。

注意，中序线索链表中有两个空指针，但先序和后序链表中不一定只有一个空指针，也可能有两个空指针（可用只有两个结点的二叉树进行验证）。

有时为了某些运算的方便，也可在线索二叉链表上附加头结点：其左指针或右指针指向根结点，另一指针指向遍历序列的首结点或尾结点。进一步，如果遍历序列的首结点或尾结点的线索为空，还可将它指向头结点，形成某种“循环”链表。

---

① n 个结点共 2n 个指针域，但树中只有 n−1 条边，对应 n−1 个非空指针，故空指针数为 2n−(n−1)=n+1。

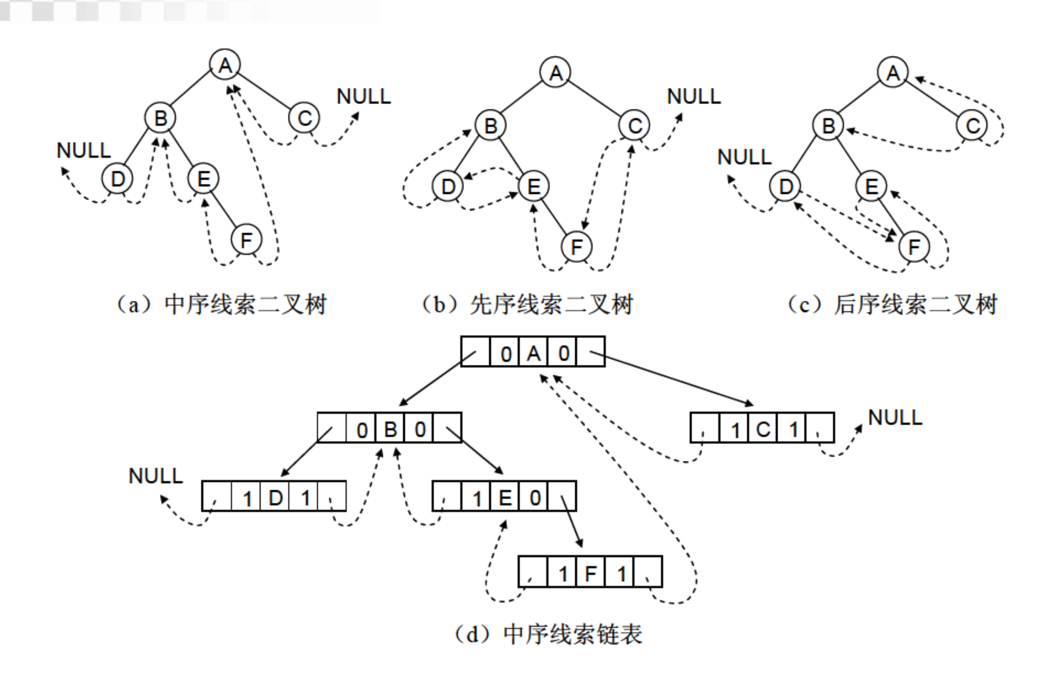


图 5.22　线索二叉树及其存储结构

以上两个线索标志也可合并为一个 tag，比如用 0、1 表示左线索的情况，用 3、4 表示右线索的情况，但不如左右分开处理简便、直观；另外，为了区分结点中的指针指的是孩子还是线索，也可不用标志位，而用负地址表示：将线索以负数的形式存放。这样如果指针域为正则指向孩子，否则表示线索。这个方法不改变结点的存储结构，也不耗费额外的空间，但指针类型毕竟不是整数类型，在使用时需要进行类型转换，这会增加运行时间。

将二叉树变为线索二叉树的过程称为**线索化**。按某种次序将二叉树线索化，只要按该次序遍历二叉树，在遍历过程中用线索取代空指针即可。为此，可用一个指针 pre 始终指向刚访问过的结点，用指针 p 指示当前正在访问的结点。显然，结点*pre 是结点*p 的前趋，而*p 是*pre 的后继。以中序线索化算法为例，该算法与中序遍历算法类似，区别仅在于访问根结点时所做的处理不同。在线索化算法中，访问当前根结点*p 所做的处理是：

（1）若结点*p 为空，则什么也不做。

（2）对结点*p 及其前趋结点*pre 进行相互处理：

① 若结点*p 的左子树为空，则令 p–>lchild 为指向其中序前趋结点*pre 的左线索（即 p–>lchild=pre），并置左线索标志（即 p–>ltag=1）。

② 若结点*pre 存在（即 pre!=NULL），且其右子树为空，则令 pre–>rchild 为指向其中序后继结点*p 的右线索（即 pre–>rchild=p），并置右线索标志（即 pre–>rtag=1）。

（3）将 pre 指向刚刚访问过的结点*p（即 pre=p）。这样，在下一次访问一个新结点*p 时，*pre 为其前趋结点。

下面给出线索链表的形式说明及中序线索化算法：

```
typedef struct node * pointer;       //pointer 类型用于表示树的一般结点
struct node {                        //结点结构
  datatype data;
  pointer lchild,rchild;
  int ltag,rtag;
```

```
};
typedef pointer bitree;              //bitree 类型用于表示树的根结点，即表示树
pointer pre=NULL;                    //全局量，初值为 NULL
void inthread(bitree t) {            //中序线索化
  pointer p;
  if(t==NULL) return;
  p=t;
  inthread(p->lchild);               //左子树线索化
  if(p->lchild==NULL) {p->lchild=pre;p->ltag=1;}   //对 p 建左线索
  if(pre!=NULL && pre->rchild==NULL) {pre->rchild=p;pre->rtag=1;}
                                                    //对 pre 建右线索
  pre=p;
  inthread(t->rchild);               //右子树线索化
}
```

类似地可得前序线索化和后序线索化算法（略）。

建立了线索链表之后，就可讨论线索二叉树上的运算了。下面以中序线索二叉树为例，介绍线索二叉树上两种比较简单又常用的运算。

1．查找某结点*p 的中序前趋和后继

查找结点*p 的中序后继结点分两种情形：

（1）若*p 的右线索标志为 1，则 p–>rchild 为右线索，直接指向*p 的中序后继结点。

（2）若*p 的右线索标志为 0，则*p 的中序后继要进行查找，它是*p 的右子树中序遍历的第一个结点，也就是右子树中“最左下”的结点。

查找结点*p 中序后继结点的算法如下：

```
pointer innext(bitree t) {
  pointer q;
  if(t->rtag==1) return(t->rchild);
  q=t->rchild;
  while(q->ltag==0) q=q->lchild;
  return q;
}
```

类似，查找结点*p 的中序前趋结点也分两种情形：

（1）若*p 的左线索标志为 1，则 p–>lchild 为左线索，直接指向*p 的中序前趋结点。

（2）若*p 的左线索标志为 0，则*p 的中序前趋要进行查找，它是*p 的左子树中序遍历的最后一个结点，也就是左子树中“最右下”的结点。

可见，在中序线索二叉树上找某点*p 的中序前趋和后继都比较方便。如果是非线索二叉树，则找某点的中序前趋和后继时，只能从根开始进行中序遍历。

然而，在前序线索链表中找某点的前序前趋、在后序线索链表中找某点的后序后继并不一定方便（前者找前序后继、后者找后序前趋方便）。这是因为要查找的前趋或后继可能就是给定点的双亲，或在双亲的另一棵子树中，如果线索标志为 0，就需要知道给定点的双亲，而找双亲一般要从根开始进行搜索（除非结点带双亲指针）。如果实际问题正好只需要其中比较方便的一个线索，则这类链表还是有使用价值的，这时只建一个线索就够了。有关内容就不细述了。

2．遍历线索二叉树

这里的遍历是指利用线索进行的遍历。遍历某种线索二叉树，可从该遍历次序下的开

始结点出发，反复找结点在该次序下的后继，直至终端结点（或从该次序下的终端结点出发，反复找结点在该次序下的前趋，直至开始结点）。以中序线索二叉树的中序遍历为例，算法如下：

```
intraver(bitree t) {              //遍历中序线索二叉树
  pointer p;
  if(t==NULL) return;             //空树
  p=t;
  while(p->ltag==0) p=p->lchild;//找中序序列的开始结点
  do {
    cout<<p->data<<endl;          //访问结点*p，假设为输出结点内容
    p=innext(p);                  //找*p 的中序后继结点
  } while(p!=NULL);
}
```

由于中序序列的终端结点的后继线索为空，所以 do 语句的终止条件是 p=NULL。显然该算法的时间复杂度仍为 O(n)，但常数因子比非线索遍历算法小，且无需设立递归栈。因此，若经常要对二叉树遍历、查找结点或查找结点在指定次序下的前趋和后继等，采用线索二叉树较好。

注意，对前序线索二叉树，找前序前趋不方便，所以遍历时应从开始结点（最左下结点）出发，按找前序后继的方式进行，否则并不比直接遍历非线索二叉树有优势。类似，对后序线索二叉树，找后序后继不方便，遍历时应从终端结点（根）开始，按找后序前趋的方式进行。

上面介绍的两种运算，线索树均优于非线索树。但如果要在线索二叉树中进行插入和删除运算就不方便了，因为这时除了修改指针外，还要修改相应的线索，运算量几乎与重新进行线索化相当。这部分内容就不介绍了。

5.7 树和森林

本节将建立树、森林和二叉树的对应关系，并讨论树的存储表示及其遍历。

5.7.1 树、森林与二叉树的转换

在树或森林与二叉树之间有一个自然的一一对应关系。任何一个森林或一棵树都可唯一地对应到一棵二叉树；反之，任何一棵二叉树也能唯一地对应到一个森林或一棵树。这样，对树或森林的一些操作就可利用二叉树来实现。

1. 树、森林到二叉树的转换

树中每个结点可能有多个孩子，但二叉树中每个结点最多只能有两个孩子。要把树转换为二叉树，就必须找到一种结点与结点之间最多 1 对 2 的关系。注意到树中的每个结点最多只有一个最左边的孩子（长子）和一个右邻的兄弟，据此我们就能很自然地将树转换成二叉树，即将各结点的长子变成其左孩子，右兄弟变成其右孩子，如图 5.23（a）所示。转换后，二叉树中左分支上相邻的结点在原树中是父子关系；右分支上相邻的结点在原树

中是兄弟关系。由于根结点没有兄弟，所以树转化后二叉树的根结点没有右子树（为空）。

这里要指出，对于无序树，结点的孩子不分左右，于是谁是长子、谁是右兄弟以及整个转换就变得不确定或不唯一。为此我们约定，对一棵无序树，在转换时，就其当前形态按有序树进行处理。易见，这样转换的结果是唯一的。具体转换时可按如下步骤进行：

（1）在所有兄弟结点之间加一连线。

（2）对每个结点，除了保留与其长子的连线外，去掉该结点与其他孩子的连线。

使用上述变换法，图 5.23（b）所示的树就变为图 5.23（c）的形式，它已是一棵二叉树，若将它按顺时针方向旋转约 45° 就能更清楚地变为图 5.23（d）所示的二叉树。

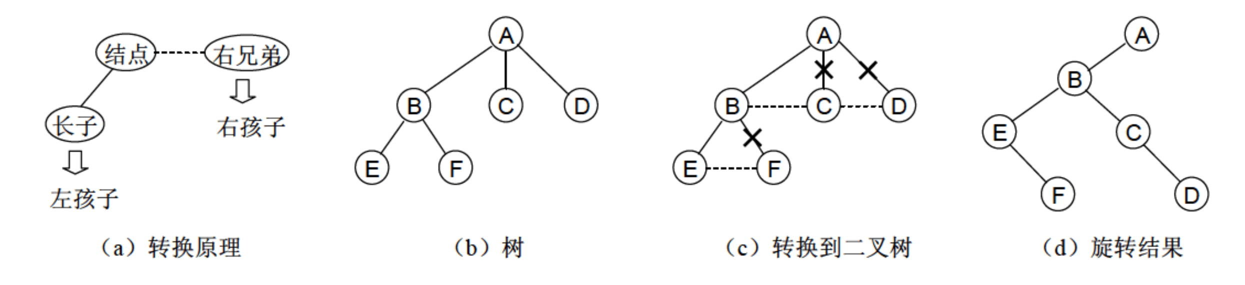


图 5.23　树转换为二叉树

将一个森林转换为二叉树的方法是，先将森林中的每一棵树变为二叉树，然后将各二叉树的根结点视为兄弟连在一起。注意，森林转化后二叉树的根结点有右子树（m>0 时）。例如在图 5.24 中，子图（b）是子图（a）的转换结果，子图（c）是子图（b）旋转后的二叉树。

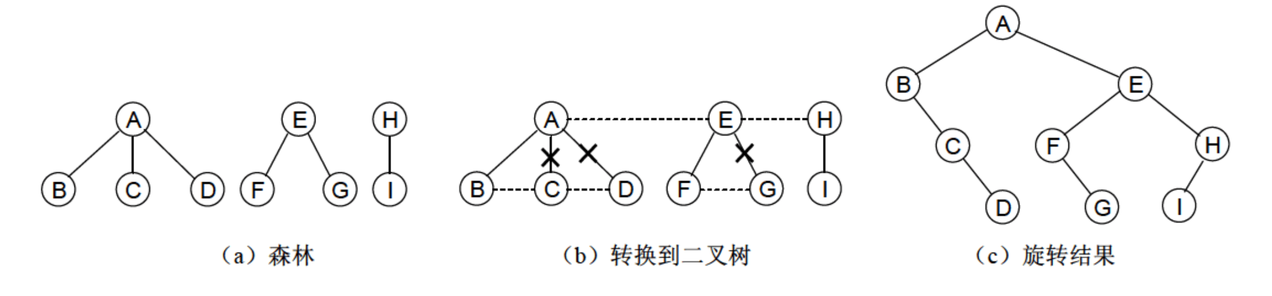


图 5.24　森林转换为二叉树

可以给上述转换方法做如下形式定义：设 $F=\{T_1, T_2, \cdots, T_m\}$表示由树 T_1、T_2、…、T_m 组成的森林，则森林 F 对应的二叉树 B(F)为：

（1）若 F 为空（m=0），则 B(F)为空二叉树。

（2）若 F 非空（m>0），则 B(F)的根就是 T_1 的根；B(F)的左子树是由 T_1 去掉根后的子树森林 $F_1=\{T_{11}, T_{12}, \cdots, T_{1m'}\}$转换成的二叉树 $B(F_1)$；B(F)的右子树是由森林 $F'=\{T_2, T_3, \cdots, T_m\}$转换成的二叉树 B(F′)。由于左子树 $B(F_1)$和右子树 B(F′)本身又要由森林转换而来，故整个转换过程需要递归地进行。

2．二叉树到树、森林的转换

上述树、森林到二叉树的转换过程是可逆的，所以反过来便可将二叉树转换到树或森林，即将二叉树中结点的左孩子变为其长子，右孩子变为其右兄弟。具体过程可如下

进行：

（1）若某结点是其双亲的左孩子，则把该结点的右孩子、右孩子的右孩子、……，都与该结点的双亲用连线连起来。

（2）去掉原二叉树中所有双亲到右孩子的连线。

图 5.25（c）就是用这种方法将图 5.25（a）所示的二叉树处理后的结果，由于原二叉树有右子树，所以最后得到的是森林。

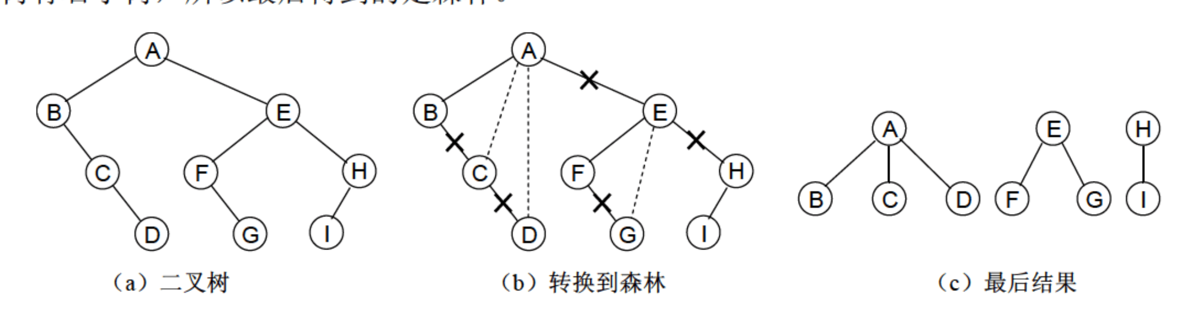


图 5.25　二叉树转换为森林

5.7.2　树的存储

对树而言，一般不能采用顺序存储方式，只能采用链式结构，即除了存储各结点本身的数据外，还要用指针表示结点间的逻辑关系（父子关系）。下面介绍树的 3 种常用存储方法，每种方法中各结点的结构相同。

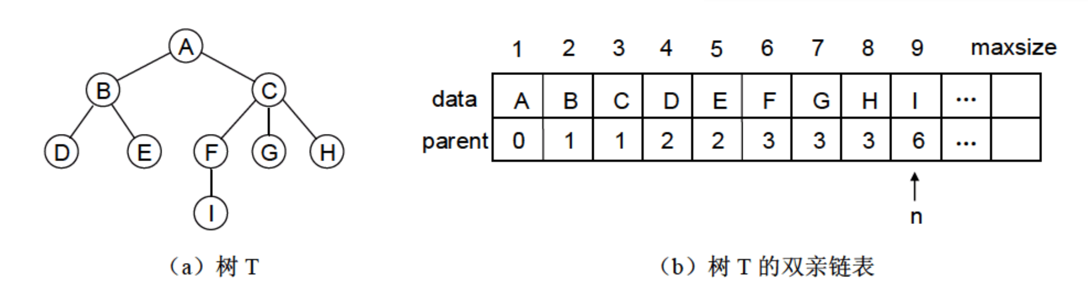
1．双亲链表表示法

该方法就是在存储结点信息的同时，为每个结点附设一个指向其双亲的指针 parent。尽管可用动态链表来实现这种表示法，然而用静态链表更为方便，其类型定义如下：

```
const int maxsize=100;      //结点数的最大值，假设为 100
typedef struct {
  datatype data;            //数据域
  int parent;               //双亲域(静态指针域)
} pnode;
typedef struct {
  pnode nodes[maxsize+1];   //结点数组，0 号单元未用
  int n;                    //结点数
} ptree;                    //静态双亲链表类型
ptree P;                    //静态双亲链表
```

各个结点在结点数组中的存储顺序原则上是任意的，但一般将根放在开始位置（否则根需要查找），并且常常按层序编号的顺序存放。这里数组下标从 1 开始使用是为了运算上的方便：编号为 i（编号从 1 开始）的结点就直接存放到 i 号单元。但 0 号单元也可用来存放其他信息，如结点总数等（这时 ptree 类型中的数据域 n 可省略）。

在双亲链表中，若结点 i 的双亲是结点 j，则 nodes[i].parent=j，若结点 i 是根结点，则 nodes[i].parent=0。对图 5.26（a）所示的树 T，其双亲链表见图 5.26（b）。



（a）树 T　　（b）树 T 的双亲链表

图 5.26　树的双亲链表表示示意图

显然，在这种存储方式下，找指定结点的双亲或祖先（包括根）十分方便，只要沿着结点的双亲指针搜索即可。但若求指定结点的孩子或子孙，则可能要遍历整个数组。另外，求结点的兄弟也不方便。不过，若结点按层序编号顺序存储，则孩子的下标大于双亲的下标，兄弟结点间的下标从左向右递增，利用这些特点，可在遍历搜索中节省一定的时间。以求结点 i 的长子为例，算法如下：

```
int firstchild(ptree *P,int i) {
  int j;
  for(j=i+1;j<=P->n;j++)
    if(P->nodes[j].parent==i) return j;//第一个双亲为 P[i]的结点是 P[i]的长子
  return 0;                             //未找到
}
```

2．孩子链表表示法

树中各个结点的孩子个数一般不同，如果采用类似二叉链表的方法来表示结点及其孩子的关系，则每个结点设置多少个孩子指针域难以确定。若以树的度 k 来设置，则 n 个结点的树中，其空指针域的数目是 $kn-(n-1)=n(k-1)+1$，这将造成极大的空间浪费。若每个结点按其实际孩子数设置指针，就需要在结点内设置度数域 degree 以指出实际指针数，这样一来，各结点不等长也不同构，且插入、删除时结点大小还需重新调整，使用不便。

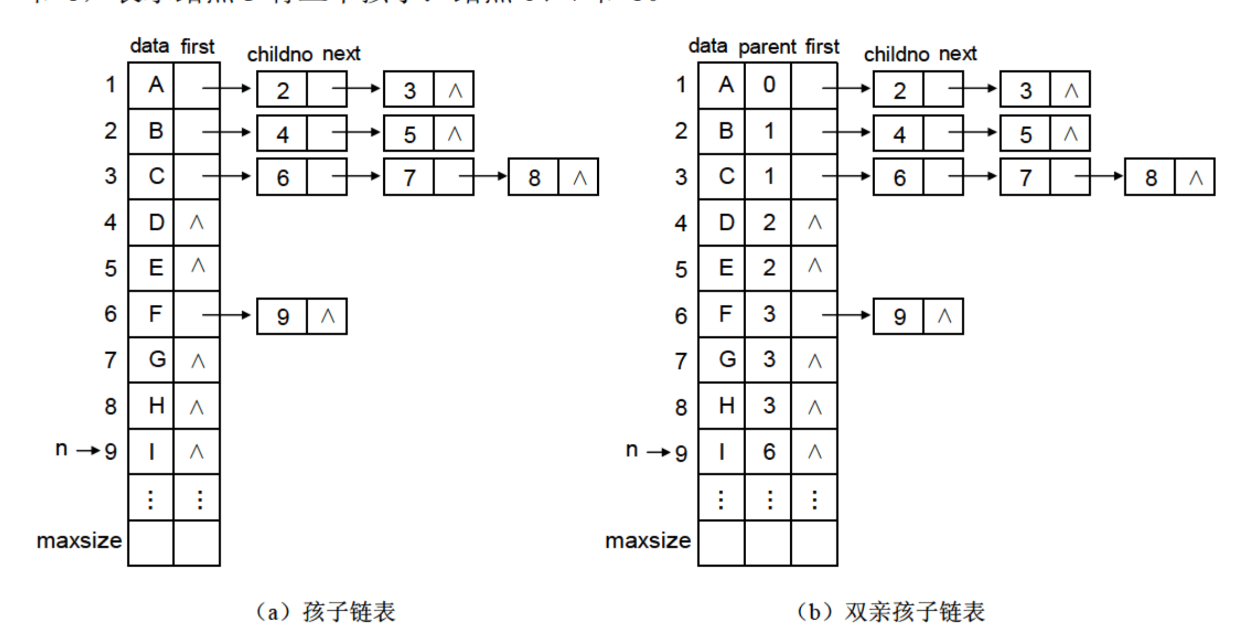
一种比较好的方法是，为树中每个结点各自建立一个孩子链表。每个孩子链表由其头指针唯一确定，为了便于查找，将所有头指针用一个数组集中存放，并与存放结点数据的结点数组合并成一个结构数组，称为表头数组，即数组中每个元素由两部分组成：数据域和指针域。其中，数据域存放结点的内容信息，指针域存放该结点的孩子链表的头指针。其类型定义如下：

```
const int maxsize=100;           //结点数的最大值，假设为 100
typedef struct cnode * pointer; //孩子链表结点指针类型
struct cnode {                   //孩子链结点结构
  int childno;                   //孩子结点的序号
  pointer next;
};
typedef struct {
  datatype data;
  pointer first;
} hnode;                         //表头结点类型
typedef struct {
  hnode nodes[maxsize+1];        //表头数组，0 号单元未用
  int n;                         //结点数
```

```
} childlink;                       //孩子链表类型
childlink T;                       //孩子链表
```

这里表头数组也从 1 号单元开始使用（0 号单元可用来存放其他信息，如根结点位置）。各结点信息在表头数组中一般也按层序编号的顺序存放，这时 1 号单元就是根结点；否则，宜在链表类型中增加一个根结点位置指针，因为在这种链表中找根结点不方便。

图 5.27（a）就是图 5.26（a）所示树的孩子链表表示。其中，若某结点 i 为叶子，则其孩子链表为空，即 nodes[i].first=NULL。对非叶子结点，nodes[i].first!=NULL，对应的孩子链表由结点 i 的所有孩子结点的序号构成。如 nodes[3].first 所指的链表有三个结点 6、7 和 8，表示结点 3 有三个孩子：结点 6、7 和 8。



（a）孩子链表　　　　（b）双亲孩子链表

图 5.27　树的孩子链表表示示意图

与双亲链表表示法相反，孩子链表表示便于实现涉及孩子和子孙的运算，但不便于实现与双亲有关的运算。因此可将这两种表示法结合起来，形成双亲孩子链表表示法。图 5.27（b）就是图 5.26（a）所示树的双亲孩子链表表示。

3. 孩子兄弟链表表示法

在介绍树转换为二叉树时，实际上利用和证明了这样一个事实：树中结点之间的关系，可用该结点与其最左边的孩子和右邻的兄弟之间的关系来描述。因此，在存储各结点信息的同时，附加两个指针域，分别指向该结点最左边的孩子和右邻的兄弟，就得到树的孩子兄弟链表表示。这其实相当于存储与树对应的二叉树。例如，图 5.26（a）所示树的孩子兄弟链表如图 5.28 所示。

这种存储结构的最大优点是，它和二叉树的二叉链表表示完全一样，只是结点的两个指针的含义有所不同。因此，可利用二叉树的算法来实现对树的操作。

以上方法中逻辑信息是显式表示的，其中指针的空间开销较大。树还有一些“压缩”型的储存方法，逻辑信息用占空间很小的一些相关信息（如左、右孩子标志，结点度数等），并结合特定的存放顺序（如先根、后根、层次顺序等）间接表示，在具体使用时再由这些相关信息求出逻辑信息，即以时间换空间，相对比较烦琐，这里从略。

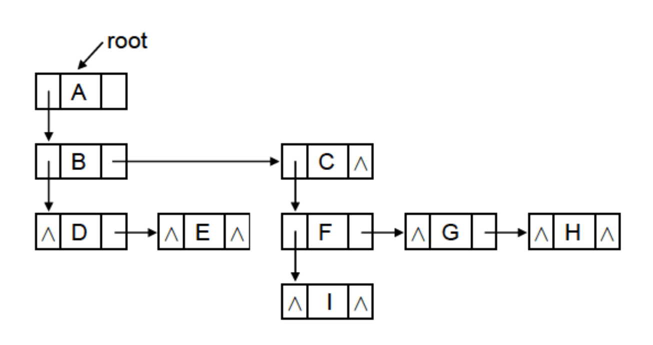


图 5.28　树的孩子兄弟链表表示示意图

5.7.3　树和森林的遍历

在树和森林中，每个结点可有两棵以上的子树，并且各结点子树的个数也不一定相同，因此不便讨论它们中根次序的遍历，但可研究先根和后根次序的遍历。设树 T 如图 5.29 所示，结点 R 为根，根的子树从左到右依次为 T_1、T_2、…、T_k。树的两种遍历方法定义为：

（1）前序遍历树 T

若树 T 非空，则：

① 访问根结点 R。

② 依次前序遍历根 R 的各子树 T_1、T_2、…、T_k。

（2）后序遍历树 T

若树 T 非空，则：

① 依次后序遍历根 R 的各子树 T_1、T_2、…、T_k。

② 访问根结点 R。

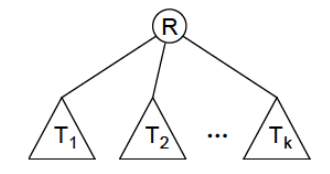


图 5.29　树 T

例如，对图 5.30（a）所示的树进行前序遍历和后序遍历，得到的前序序列和后序序列分别是 ABECD 和 EBCDA。

值得注意的是：前序遍历一棵树恰好等价于前序遍历该树对应的二叉树，后序遍历一棵树恰好等价于中序遍历该树对应的二叉树。这可由树与二叉树的转换关系以及树与二叉树遍历的定义推得。例如在图 5.30 中，子图（b）是子图（a）对应的二叉树，它的前序序列和中序序列正是 ABECD 和 EBCDA。

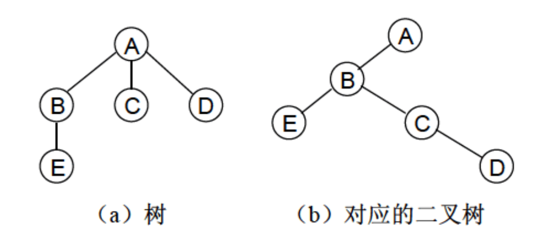


图 5.30　树和对应的二叉树

类似地，可得到森林的两种遍历方法：

（1）前序遍历森林

若森林非空，则：

① 访问森林中第一棵树的根结点。

② 前序遍历第一棵树中根结点的各子树所构成的森林。

③ 前序遍历除第一棵树外其他树构成的森林。

（2）后序遍历森林[①]

若森林非空，则：

① 后序遍历森林中第一棵树的根结点的各子树所构成的森林。

② 访问第一棵树的根结点。

③ 后序遍历除第一棵树外其他树构成的森林。

简言之，前序遍历森林就是从左到右依次前序遍历森林中的每一棵树，后序遍历森林就是从左到右依次后序遍历森林中的每一棵树。

和遍历树类似，前序遍历森林等价于前序遍历该森林对应的二叉树，后序遍历森林等价于中序遍历该森林对应的二叉树。这同样可由森林与二叉树的转换关系以及森林与二叉树遍历的定义推得。例如，对图 5.31（a）所示的森林进行前序遍历和后序遍历，得到该森林的前序序列和后序序列分别为 ABECDFGH 和 EBCDAGHF。图 5.31（b）是该森林对应的二叉树，它的前序序列和中序序列也分别为 ABECDFGH 和 EBCDAGHF。

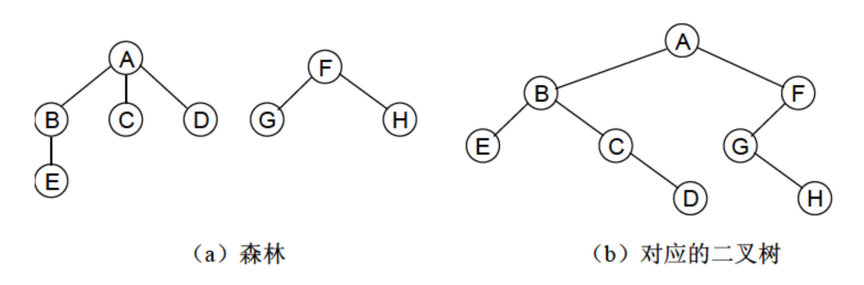


图 5.31　森林和对应的二叉树

由上述讨论可知，当用二叉链表作为树和森林的存储结构时，树和森林的前序遍历和后序遍历，可用二叉树的前序遍历和中序遍历算法来实现。另外，对树和森林也可以进行层序遍历。

（1）层序遍历树 T

若树 T 非空，则：

① 访问根结点 R。

② 若第 i（i≥1）层结点已访问，则访问第 i+1 层结点时，按从左到右的次序依次访问第 i+1 层上的结点。

显然，对树按层序遍历得到的遍历序列与结点的层序编号一致。事实上，对结点进行层序编号的过程本身就是一种层序遍历（访问结点的操作就是给它编个号）。

（2）层序遍历森林

这就是将森林中各树的同层结点依次输出。例如，图 5.31（a）所示森林的层序遍历序列为 AFBCDGHE。

注意，层序遍历森林不是依次对森林中的每一棵树进行层序遍历，而是所有树按同层结点依次输出。这不难理解：设想有一个虚的（总）根结点，使森林中的各树作为其子树，

---

① 有的文献称此为森林的中序遍历，即按第一棵树的根（D）、第一棵树的子树森林（A）、第一棵树外其他树构成的森林（B）三者访问的先后顺序来定义森林的遍历：先根 DAB、中根 ADB、后根 ABD。

则森林的层序遍历就是该树的层序遍历，也即各子树按同层结点依次输出。

易见，层序遍历树或森林，等价于沿右指针方向“层序”遍历对应的二叉树。这里，在对应二叉树中，结点与其右孩子，右孩子的右孩子，……，为同层结点，而左指针起到承上启下的作用。据此不难写出具体的层序遍历算法（略，见习题 5.21）。

5.8 哈夫曼树及其应用

树的应用非常之广，本节以**哈夫曼**（Huffman）树为例介绍树的应用。

5.8.1 最优二叉树（哈夫曼树）

在许多应用中，常常将树中结点赋予一个有某种意义的实数，称为该结点的权。结点到树根之间的路径长度与该结点权的乘积称为该结点的带权路径长度。树的（外部）**带权路径长度**（Weighted Path Length），定义为树中所有叶子结点的带权路径长度之和，通常记为：

$$WPL=\sum_{i=1}^{n} w_i l_i$$

其中，n 表示叶子结点的数目，w_i 和 l_i 分别表示叶结点 i 的权值和它到根之间的路径长度。在权为 w_1、w_2、…、w_n 的 n 个叶结点的所有二叉树中，带权路径长度 WPL 最小的二叉树称为**最优二叉树**或**哈夫曼树**。

在给定了叶子及其权后，如何构造最优二叉树呢？先看一个例子。

给定 4 个叶子结点 a、b、c 和 d，权分别为 6、3、4 和 8。我们可以构造如图 5.32 所示的 3 棵二叉树（还有多个）。它们的带权路径长度分别为：

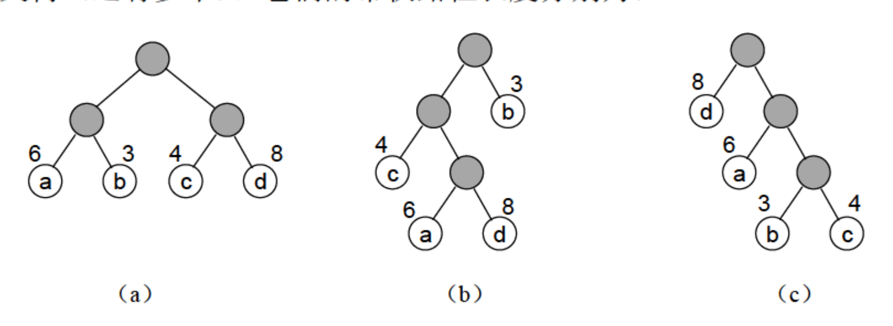


图 5.32 叶子相同的三棵二叉树

（a）WPL=6×2+3×2+4×2+8×2=42

（b）WPL=6×3+8×3+4×2+3×1=53

（c）WPL=3×3+4×3+6×2+8×1=41

其中树（a）为完全二叉树，但它的 WPL 并不是最小；最小的是树（c），下面将看到，它就是哈夫曼树。树（c）有个特点：权越大的叶子离根越近。这可从 WPL 的表达式来理解：如果权 w_i 较大，则希望路径 l_i 较小，这样 $w_i l_i$ 就会较小，从而有利于减小 WPL。不过

这种分析是不严格的，因为 WPL 考虑的是总体情况，而不是个别叶子带权路径的变长或变短，但一般情况下这个特点还是存在的。在此基础上，1952 年哈夫曼给出了一个求最优二叉树的精巧方法，称之为哈夫曼算法，步骤如下：

（1）首先，根据给定的 n 个权值 w_1、w_2、…、w_n 构造含有 n 棵二叉树的森林 F={T_1, T_2, …, T_n}，其中每棵二叉树 T_i 中只有一个权值为 w_i 的根结点，没有左右子树。

（2）在森林 F 中选出两棵根结点权值最小的树（当这样的树不止两棵时，可从中任选两棵），将这两棵树合并成一棵新的二叉树。这时会增加一个新的根结点，它的权取为原来两棵树的根的权值之和，而原来的两棵树就作为它的左右子树（谁左、谁右无关紧要）。

（3）对新的森林 F 重复步骤（2），直到森林 F 中只剩下一棵树为止。这棵树便是哈夫曼树。

按照这个算法，权越大的叶子合并的时机越晚，它离最终的根也就越近，正好符合上面的定性分析和观察结果。

由哈夫曼算法可知，哈夫曼树不一定唯一（但 WPL 是相同的，并且都为最小）。原因在于每次合并时，原两棵树谁左谁右，以及候选的两棵树有多个时选哪两个等。图 5.33 表示了用该算法从上述 4 个叶子构造哈夫曼树的过程，其中在合并时将根权值小的二叉树作为新根的左子树，其结果（d）就是图 5.32 中的树（c）。

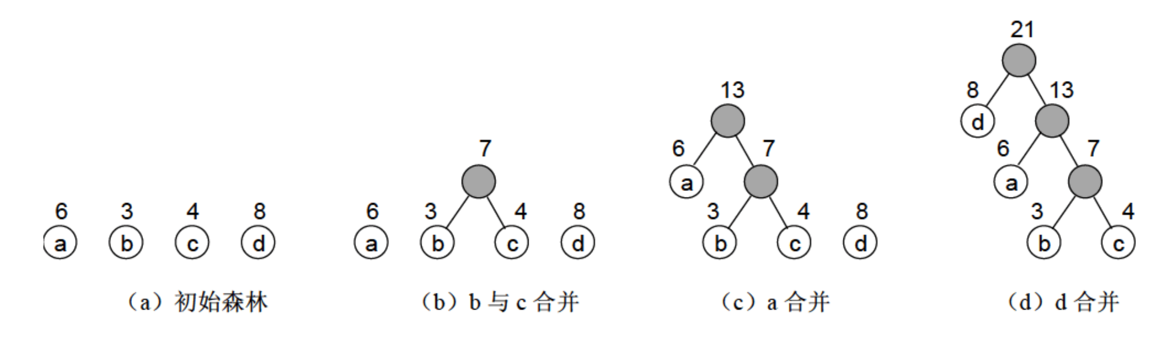


图 5.33 哈夫曼树的构造过程

在哈夫曼算法中，初始森林共有 n 棵二叉树，每棵树中仅有一个结点，它们既是根，又是叶子。算法的第二步是将当前森林中的两棵根结点权值最小的二叉树合并成一棵新二叉树。每合并一次，森林中就减少一棵树。显然，要进行 n−1 次合并，才能使森林中二叉树由 n 棵减少到只剩一棵，即最终的哈夫曼树。另外，每合并一次都要产生一个新结点，合并 n−1 次共产生 n−1 个新结点。由此可知，最终求得的哈夫曼树中共有 n+(n−1)=2n−1 个结点，其中的叶结点就是初始森林中的 n 个孤立结点。

在哈夫曼树中，每个分支结点都是合并过程中产生的，它们的度为 2，所以树中没有度为 1 的分支结点，这类树通常称为**严格二叉树**[①]（Strictly Binary Tree）或**正则二叉树**（Proper Binary Tree）。实际上所有具有 n 个叶结点的严格二叉树都恰好有 2n−1 个结点。

我们用一个大小为 2n−1 的数组来存储哈夫曼树中的结点，其存储结构为：

```
const int n=20;              //叶子结点数，假设为 20
```

① 有的文献称此为满二叉树（Full Binary Tree）；有的文献称此为完全二叉树（Complete Binary Tree）；也有其他称呼，如 2-tree、强二叉树等。

```
const int m=2*n-1;            //结点总数
typedef struct {
  float weight;
  int parent,lchild,rchild;
} nodetype;                   //结点类型
typedef nodetype hftree[m];//哈夫曼树类型，数组从 0 号单元开始使用
hftree T;                     //哈夫曼树向量
```

其中，每个结点包括 4 个域，weight 是结点的权值，lchild、rchild 分别为结点的左、右孩子在数组中的下标，叶结点的这两个指针值为−1；parent 是结点的双亲在数组中的下标。这里设置 parent 域不仅可使以后涉及双亲的运算方便，还可在合并时区分根结点和非根结点：若 parent 的值为−1，则该结点无双亲，即为根结点，尚未合并过。之所以要区分根结点与非根结点，是因为每次合并两棵二叉树时，要先在当前森林的所有结点中找两个权值最小的根结点，因此，有必要为每个结点设置一个标记以区分根结点和非根结点。

在上述存储结构上实现的哈夫曼算法可粗略地描述为：

（1）初始化：将初始森林的各根结点（叶子）的双亲和左、右孩子指针置为−1。

（2）输入叶子权：叶子在结点向量 T 的前 n 个分量中，构成初始森林的 n 个根结点。

（3）合并：对森林中的树进行 n−1 次合并，共产生 n−1 个新结点，依次放入结点向量 T 的第 i 个分量中（$n \leq i \leq m-1$）。每次合并的步骤是：

① 在当前森林的所有结点 T[j]（$0 \leq j \leq i-1$）中，选取具有最小权值和次小权值的两个根结点，分别用 p1 和 p2 记住这两个根结点在结点向量 T 中的下标。

② 将根为 T[p1]和 T[p2]的两棵树合并，使其成为新结点 T[i]的左右孩子，得到一棵以新结点 T[i]为根的二叉树。同时修改 T[p1]和 T[p2]的双亲域，使其指向新结点 T[i]，这意味着它们在当前森林中已不再是根。T[p1]和 T[p2]的权值相加后，作为新结点 T[i]的权值。

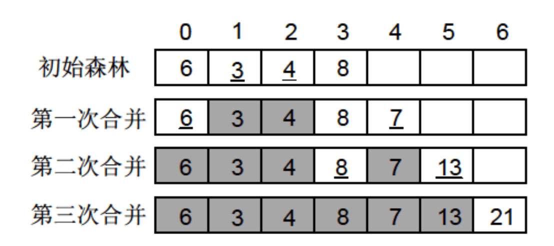
求精后的哈夫曼算法如下：

```
void huffman(hftree T) {
  int i,j,p1,p2;
  float small1,small2;
  for(i=0;i<n;i++) {                //初始化
    T[i].parent=-1;
    T[i].lchild=T[i].rchild=-1;
  }
  for(i=0;i<n;i++)                  //输入 n 个叶子的权
    cin>>T[i].weight;
  for(i=n;i<m;i++) {                //进行 n-1 次合并，产生 n-1 个新结点
    p1=p2=-1;                       //此句可不要
    small1=small2=FLOAT_MAX;        //FLOAT_MAX 为 float 类型的最大值
    for(j=0;j<=i-1;j++) {           //找两个权值最小的根结点
      if(T[j].parent!=-1) continue; //不考虑已合并过的结点
      if(T[j].weight<small1) {      //修改最小权、次小权及其位置
        small2=small1;
        small1=T[j].weight;
        p2=p1;
        p1=j;
      }
      else if(T[j].weight<small2) { //修改次小权及位置
        small2=T[j].weight;
        p2=j;
```

```
            }
          }
          T[p1].parent=T[p2].parent=i;
          T[i].parent=-1;                    //新根
          T[i].lchild=p1;
          T[i].rchild=p2;
          T[i].weight=small1+small2;
        }
      }
```

注意，哈夫曼树虽然不一定唯一，但算法 huffman 得到的结果是唯一的，因为按该算法执行时，如果中途有两个以上的根权都最小，则它选中的是数组下标较小的两个；并且在合并时，权最小的根作左子树，权次小的根作右子树，故不会出现不确定的情况。

以前面 4 个权为 6、3、4、8 的叶子为例，用上述算法求哈夫曼树的过程见图 5.34。注意，图中只画出了 weight 域的变化情况。因为 n=4，故进行 3 次合并。合并过的结点用阴影表示，以后不再考虑；当前权值最小的两个根结点用下划线表示。



	0	1	2	3	4	5	6
初始森林	6	3	4	8			
第一次合并	6	3	4	8	7		
第二次合并	6	3	4	8	7	13	
第三次合并	6	3	4	8	7	13	21

图 5.34 哈夫曼算法的执行过程

5.8.2 哈夫曼编码与压缩

哈夫曼树的应用很多，本节介绍哈夫曼编码与压缩。

我们在使用计算机的时候，为了减少数据文件所占用的磁盘空间，或者为了提高网络数据传递的时间效率，经常对文件进行压缩以减少文件的大小，如常见的压缩格式有 ZIP、LHA、RAR 等。其实，利用哈夫曼算法，也能进行简单的文件压缩。

假设文件中有 n 个字符，则该文件大小就是 n 字节（byte），或 8n 位（bit），这是因为每个字符的 ASCII 编码为 8 位，最多可以表示 $2^8=256$ 种字符。若实际使用的字符种类较少，则可不必用 8 位编码，比如只有 30 种时只需 5 位编码就可以了（$2^5=32>30$），这样可一定程度地节省文件长度。一般对 k 种字符编码需要 $\lceil \log_2 k \rceil$ 位。以上各种字符的编码长度都相等，称为**等长码**。但是，文件中各个字符使用的频率是不等的，如英文中 E 和 T 的使用一般就比 Q、X、Z 等频繁几十倍。若采用不等长编码，让使用频率高的字符的编码缩短，就可使文件总长变短。注意，不等长编码后文件应以二进制形式存放。

然而，采用不等长编码时有可能产生多义性。例如，假设 00 表示 E，01 表示 T，0001 表示 W，则当取出的编码信息串是 0001 时，就无法确定原文是 ET 还是 W。产生该问题的原因是 E 的编码与 W 的编码的开始部分（前缀）相同。因此，不等长编码中要求任一字符的编码都不是其他字符编码的前缀，这种编码叫做**前缀（编）码**。显然，等长的 ASCII 码是前缀码。

假设组成文件的字符集是 $D=\{d_1, d_2, \cdots, d_n\}$，每个字符 d_i 在文件中出现的次数是 c_i，

对应的编码长度是 l_i，则文件总长为 $\sum_{i=1}^{n} c_i l_i$ 。因此，使文件总长最小就是使 $\sum_{i=1}^{n} c_i l_i$ 取最小值。然而，我们不可能对每个文件都去统计其中每个字符具体的出现次数 c_i，但通过对大量文件的统计分析，可以得到每个字符出现的概率 p_i，则 $\sum_{i=1}^{n} p_i l_i$ 表示平均码长。显然，平均码长越小，文件的平均总长就越短。例如，设字符集 D 及其概率分布 P 为：

D={a, b, c, d, e}

P={0.12, 0.40, 0.15, 0.08, 0.25}

在该字符集 D 上的三种不同的前缀编码如图 5.35 所示，其中编码 1、编码 2 和编码 3 的平均码长分别为 3、2.2 和 2.15。可以证明编码 3 就是上述给定概率分布下的**最优前缀码**（即平均码长 $\sum_{i=1}^{n} p_i l_i$ 最小的前缀码）。

字符	概率	编码1	编码2	编码3
a	0.12	000	000	1111
b	0.40	001	11	0
c	0.15	010	01	110
d	0.08	011	001	1110
e	0.25	100	10	10

图 5.35　三种前缀码

观察表达式 $\sum_{i=1}^{n} p_i l_i$ ，如果我们取 p_i 为叶结点的权，取编码长度 l_i 为叶结点的路径长度，则平均码长 $\sum_{i=1}^{n} p_i l_i$ 最小的问题就是一个带权路径长度最小的哈夫曼树的构造问题。于是可以这样求最优前缀码：用 d_1、d_2、…、d_n 作为叶结点，用 p_1、p_2、…、p_n 作叶结点的权，构造一棵哈夫曼树；然后将哈夫曼树中每个分支结点的左分支标 0，右分支标 1，把从根到每个叶子的路径上的标号连接起来，作为该叶子所代表的字符的编码。注意，每个字符 d_i 的编码长度对应叶子的路径长度 l_i。在哈夫曼树中，没有任何叶子是其他叶子的祖先，所以每个叶结点对应的编码不可能是其他叶结点编码的前缀，即上述编码得到的是前缀码。这个过程就是**哈夫曼编码**。据此编码就可对文件进行压缩，这个过程就不细述了。

例如，对图 5.35 给出的字符集及其概率分布，用哈夫曼算法构造的哈夫曼编码树及其对应的哈夫曼编码如图 5.36 所示。

在哈夫曼树的存储结构中，每个结点都存有其双亲指针，所以在求哈夫曼编码的过程中，可以从叶结点出发，向上回溯直到根结点。具体过程是：从叶子 T[i]出发，利用双亲指针找到其双亲 T[p]；再根据 T[p]的孩子指针可以知道 T[i]是 T[p]的左孩子还是右孩子，若是左孩子，则生成代码 0，否则生成代码 1；然后以 T[p]为出发点，继续上述过程，直到根结点。

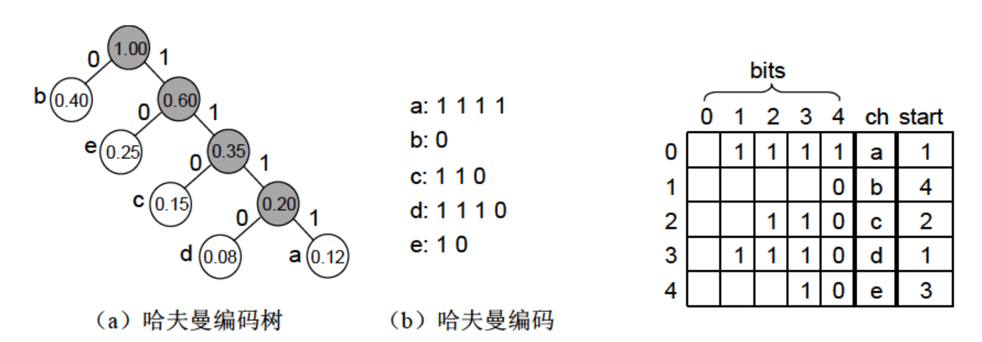
显然，这里生成的代码序列与所求编码次序相反，因此，可以将生成的代码按从后往前的次序依次存放在一个位串 bits 中。虽然各字符的编码长度不同，但最大不会超过 n，所以，bits 的大小可设为 n，然后用一个整型变量 start 来指示编码在位串 bits 中的起始位

置，其类型定义和编码算法如下：

```
typedef struct {
  char bits[n];                  //位串，存放编码
  char ch;                       //字符
  int start;                     //编码在位串中的起始位置
} nodetype;                      //编码结点类型
typedef nodetype codelist[n];    //编码表类型

void encode(codelist codes,hftree T) {
  int i,c,p,start;
  for(i=0;i<n;i++) {             //从叶结点出发向上回溯
    cin>>codes[i].ch;            //读入叶子字符
    start=n;
    c=i;
    p=T[i].parent;               //取出双亲
    while(p!=-1) {
      start--;
      if(T[p].lchild==c)  codes[i].bits[start]='0';//左子树编'0'
      else                codes[i].bits[start]='1';//右子树编'1'
      c=p;
      p=T[p].parent;
    }
    codes[i].start=start;
  }
}
```

以图 5.36（a）所示的哈夫曼树为例，上述算法求出的哈夫曼编码见图 5.37。



	0	1	2	3	4	ch	start
0		1	1	1	1	a	1
1					0	b	4
2			1	1	0	c	2
3		1	1	1	0	d	1
4				1	0	e	3

（a）哈夫曼编码树　（b）哈夫曼编码

图 5.36　哈夫曼编码树及其编码　　**图 5.37　哈夫曼编码的存储结构**

哈夫曼树也可用来译码和解压缩。与编码过程相反，译码过程是从哈夫曼树的根结点出发，逐位读入编码信息，若为 0，则走向左孩子，否则走向右孩子，一旦到达叶结点 T[i] 便译出相应的字符 codes[i].ch。然后，重新从根结点出发继续译码，直到编码压缩文件结束。为简单起见，这里假设编码信息以十进制数字形式逐位输入，则译码算法如下：

```
void decode(codelist codes,hftree T) {
  int i,b,endflag;
  endflag=-1;                    //编码结束标志，假设取-1
  i=m-1;                         //从根结点开始搜索
  while(cin>>b,b!=endflag) {     //读入一个编码位，若不是结束标志则循环
    if(b==0)  i=T[i].lchild;     //根据编码位走向左孩子或右孩子
    else      i=T[i].rchild;
    if(T[i].lchild==-1) {        //T[i]为叶子
```

```
        cout<<codes[i].ch;          //译码，输出字符
        i=m-1;                      //回到根结点开始下一次搜索
      }
    }
    if(i!=m-1) cout<<"编码有错！\n";    //编码读完，但结束点不是根，出错
  }
```

由于哈夫曼树中没有度为 1 的结点，故上面算法中仅用 T[i].lchild=−1 来判定 T[i]是否为叶结点。另外，正确的结束点是译完最后一个字符后输入−1，此时搜索点为根结点（i=m−1）。

需要指出，哈夫曼编码/解码的主要意义在于最优编码，用于文件压缩则意义不大，因为有更高压缩率的方法（如 ZIP、RAR 等，它们压缩原理不同）。

5.8.3　分类与判定树

本节介绍哈夫曼树的另一个应用：描述快速分类过程。

分类是一种常用运算，其作用是将输入的数据按预定的标准划分成不同的种类。例如，一般来说，工厂生产的产品，质量有高有低，需要根据一定的质量指标对产品进行检测，根据检测的结果将产品划分成不同的等级，这就是一个分类问题。分类问题有时也称判定问题，其中的每次检测也称一次判断。

在分类过程中，对每一次判断，产生两个结果：要么成立（真），要么不成立（假），正好可用二叉树的两个分支来表示。每次判断后产生两个子类，如果某个子类不是所要的最终结果，再对子类继续进行判断，以划分出更细的子类，直到每个子类都对应唯一的一种分类结果。于是整个分类过程可以用一棵二叉树来描述，称之为分类问题的**判定树**。在判定树中，每个分支结点对应一次判断，每个叶子结点对应一种分类结果；根对应整个分类过程的第一次判断。

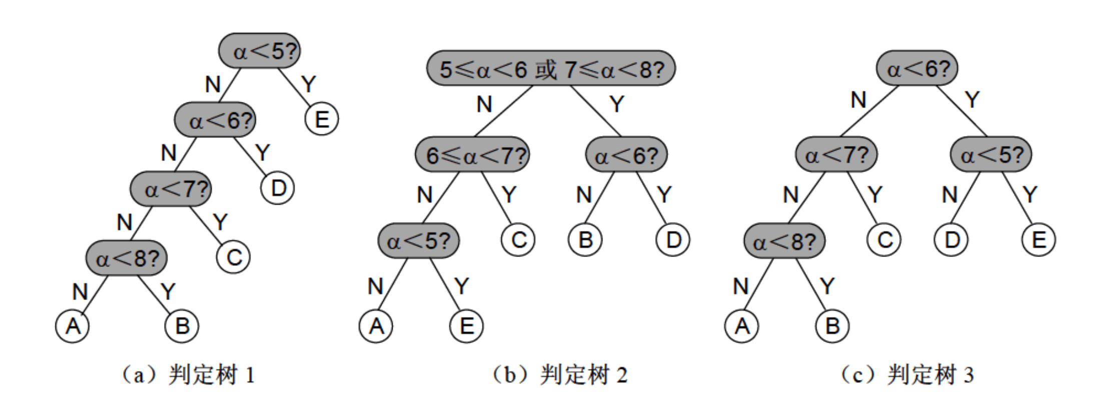
例如，假设某产品的等级标准见图 5.38。图 5.39（a）～图 5.39（c）就是该质量检测问题的三棵判定树，树上的 4 个分支结点对应于 4 次条件判断；树上的 5 个叶子结点对应于分类的 5 种不同的结果，即区分出的 5 个质量等级。

等级	E	D	C	B	A
检测值α	$\alpha<5$	$5\leq\alpha<6$	$6\leq\alpha<7$	$7\leq\alpha<8$	$8\leq\alpha$
百分比	0.15	0.2	0.35	0.2	0.1

图 5.38　质量等级标准

由于判定树描述的就是分类方法，因而由判定树很容易写出相应的分类算法。以图 5.39（a）的判定树 1 为例，分类算法如下：

```
char classify(float x) {
  if(x<5) return 'E';
  else if(x<6) return 'D';
  else if(x<7) return 'C';
  else if(x<8) return 'B';
  else return 'A';
}
```



（a）判定树 1　　（b）判定树 2　　（c）判定树 3

图 5.39　判定树示例

显然，在分类中，不同等级的产品所经过的条件判断次数一般是不同的，如按上述算法，E 类产品 1 次判断就可确定，D 类产品要判断 2 次等。另外，如果判定树不同，即使是同一类产品，分类时所经过的判断次数一般也不相同。例如，图 5.39 中的三棵判定树，除 D 类产品都需 2 次判断外，其他各类产品所需的判断次数就不完全相同了。

如果只对个别或少数产品进行分类，则采用该问题的任何一个判定树都可以。但实际中经常需要对大量的产品进行分类，如一个工厂一个月的某零件产品就可能成千上万，这时就要考虑算法的时间性能了。在分类中，主要的操作是条件判断，故以条件判断的次数为标准来分析算法的时间复杂度。

仍以上述产品质量分级问题为例。假设要分类的产品有 N 件，其中每类产品所占的比率已预测出来，见图 5.38 的第三行。设产品需要分成 n 类，每类所占的比率为 p_i（该类产品件数为 Np_i），对该类产品分类所需要的判断次数为 c_i 次，则总的判断次数为：

$$SUM = \sum_{i=1}^{n} Np_i c_i = N\sum_{i=1}^{n} p_i c_i$$

如果使上式达到最小值，则分类算法的时间性能最好。从上式看到，如果将产品所占的比率 p_i 看成是 n 个叶子的权，产品的判断次数 c_i 看成叶子的路径长度，则这个问题就是最优二叉树的构造问题。于是可利用哈夫曼算法构造哈夫曼树，从而得到最佳判定过程。具体方法是，以每个类别的概率为权构造哈夫曼树；在每次合并产生的新结点上加上适当的判定条件。图 5.39（b）就是这样的一棵判定树。

然而，图 5.39（b）虽然“判断次数”最少，但实际执行效果并不好。因为它的每次判断有些不是最基本的，如条件 $6 \le \alpha < 7$，它实际包含 2 次基本比较：$\alpha \ge 6$ && $\alpha < 7$，故总的“基本比较次数”不一定是最小的。

为此，需要对哈夫曼算法略作修改，使得树中的分支结点，即每次判断为一次基本比较。注意到对检测参数做基本比较时，以检测值为界，所划分出的两个区间（子类）在该参数空间上是“相邻”的（一分为二，两者紧邻，如区间 $\alpha<7$ 和区间 $\alpha \ge 7$），故在哈夫曼算法的每次合并时，不能简单地找两个权最小的根，还要求它们对应的空间范围“相邻”，即合并的条件是：在相邻的两个根中找权值和最小的。

对图 5.38，最初权最小的两个根是 A 和 E，但它们不相邻（中间隔着 B、C、D 等区段），故不合并；相邻的权值和最小的两个根是 A 和 B，故对它们合并。A、B 合并后形成

区间$\overline{BA}$，它和 C 相邻。第二次合并时找到的两个根是$\overline{BA}$和 C，它们合并后形成区间$\overline{CBA}$，和 D 相邻。第三次合并时找到的两个根是 E 和 D，它们合并后形成区间$\overline{ED}$，和$\overline{CBA}$相邻。最后是根$\overline{ED}$和$\overline{CBA}$合并，最终结果见图 5.39 的判定树 3。这时：

$$SUM = N\sum_{i=1}^{n} p_i c_i = N(0.15\times2+0.2\times2+0.35\times2+0.2\times3+0.1\times3) = 2.3N$$

而采用判定树 1 时：

$$SUM' = N\sum_{i=1}^{n} p_i c_i = N(0.1\times4+0.2\times4+0.35\times3+0.2\times2+0.15\times1) = 2.8N$$

可见，采用最优判定树可使比较次数下降 2.8N−2.3N=0.5N 次，节省了 17.9%，如果 N 很大，节省的比较次数和时间就相当可观了。

最后，与图 5.39 的判定树 3 对应的判定算法如下：

```
char classify3(float x) {
  if(x<6)
    if(x<5) return 'E';
    else    return 'D';
  else if(x<7) return 'C';
  else if(x<8) return 'B';
  else return 'A';
}
```

习　题　五

5.1　树的度是指树中结点的最大度数，但二叉树的度就一定为 2 吗？

5.2　已知某三叉树中度为 1、2、3 的结点数分别为 n_1、n_2、n_3，求其中的叶子结点数。

5.3　满 k 叉树有 n 个结点，则其中叶子结点有多少？

5.4　结点数为 n 高度也为 n 的二叉树是否只有左单枝和右单枝两种？

5.5　试回答下列问题：

（1）只有 3 个结点的树和二叉树各有几种不同的形态？各有多少棵不同的树？

（2）n 个结点的树和二叉树各有几种不同的形态？

5.6　找出所有满足下面条件的二叉树：

（1）先序序列和中序序列相同。

（2）后序序列和中序序列相同。

（3）先序序列和后序序列相同。

（4）先序、中序、后序序列都相同。

（5）先序序列和后序序列相反。

5.7　试回答下列问题：

（1）中序序列的最后一个结点是否就是先序序列的最后一个结点？

（2）怎样输出逆序的后根序列？

5.8　证明：

（1）完全二叉树的叶子数为 $n_0=n-\lfloor n/2\rfloor=\lceil n/2\rceil$。

（2）严格二叉树的叶子数为 $n_0=(n+1)/2$。

即这两种二叉树中叶子和非叶子结点各有一半左右。

5.9 证明：

（1）含 n 个叶子的完全二叉树的结点总数为 2n−1 或 2n。

（2）含 n 个叶子的完全二叉树的深度为 $\lceil \log_2 n\rceil+1$ 或 $\lfloor \log_2 n\rfloor+2$（它们不一定相等）。

（3）含 n 个叶子的二叉树的深度至少为 $\lceil \log_2 n\rceil+1$。

5.10 如果某完全二叉树的叶子数正好为 2 的幂次，如 $n=2^{k-1}$，则该完全二叉树的深度就是 k 吗？

5.11 证明：对非空树，$n=\sum_{i=1}^{n} D(i)+1$，即结点数=所有结点的度数之和+1。

5.12 证明：若森林中有 n 个分支结点，则对应二叉树中有 n+1 个结点没有右孩子。

5.13 已知二叉树的层序序列为 ABCDEFGHIJ，中序序列为 DBGEHJACIF，请画出该二叉树。

5.14 已知某森林的先根序列为 ABCDEFG，后根序列为 BCAFEGD，请画出该森林。

5.15 线索二叉链表就是用结点的空指针域来存放某种遍历的前趋和后继线索，是否线索二叉链表中就没有空指针了？如果有，有几个空指针？

5.16 试编写递归算法，在二叉链表上实现以下功能：

（1）求二叉树的高度。

（2）求二叉树中度为 1 的结点数。

（3）查找值为 x 的结点，若找到则返回该结点的地址；否则返回空指针。

5.17 试编写非递归算法，在二叉链表上实现以下功能：

（1）将二叉树各结点的左右子树交换。

（2）求二叉树中叶子结点数。

5.18 试编写算法，在二叉链表上实现以下功能：

（1）求根*t 到任一给定结点*s 的路径（或，求任一给定结点*s 的祖先）。

（2）求任意两点*p 和*q 的共同祖先。

5.19 试编写算法实现以下功能：

（1）由二叉树的顺序存储结构 A[1..n]求叶子结点数。

（2）由二叉树的顺序存储结构 A[1..n]建立相应的二叉链表。

（3）由二叉链表建立相应的顺序存储结构 A[1..n]。

5.20 试编写算法，在二叉链表上实现以下功能：

（1）判断二叉树是否为完全二叉树。

（2）*求二叉树的宽度。

（3）*求二叉树的高度、每个结点的层数。

5.21 假设森林（或树）用对应的二叉链表存储，试编写以下算法：

（1）求叶子数、高度。

（2）层序遍历。

5.22 能否用二叉树表示父子、兄弟、夫妻三种关系？

5.23 试编写一个将百分制转换为五分制的算法，要求平均比较次数尽可能少。假定学生成绩分布如下：

等级	E	D	C	B	A
分数	0～59	60～69	70～79	80～89	90～100
百分比	0.05	0.15	0.40	0.30	0.10

5.24 已知两个各有 m 和 n 个记录的有序文件可在 O(m+n)的移动次数内合并为一个有 m+n 个记录的有序文件。现有 5 个文件要合并为一个大文件，其中记录数分别有 20、30、10、5 和 30 个。试给出记录移动次数最少的合并步骤。

第6章 图

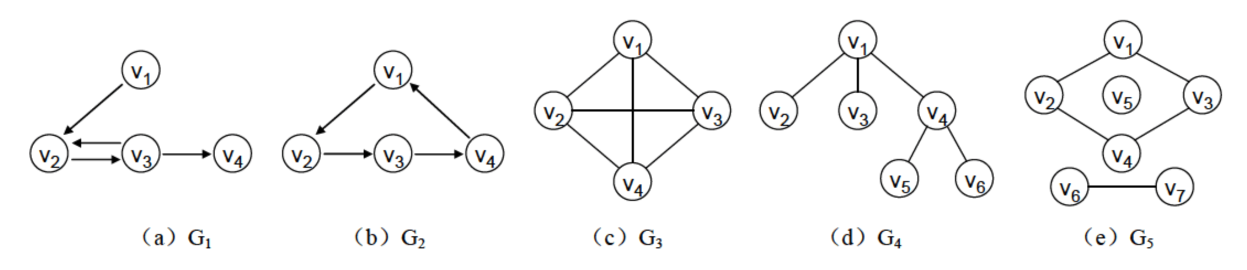
图是一种比树形结构更复杂的非线性结构。在树形结构中，结点间具有分支层次关系：除根结点外，每个结点只能和其上一层的一个结点相关；除叶子结点外，每个结点可以和下一层的多个结点相关，即“单前趋多后继”。而在图结构中，结点之间的关系是任意的，任何两结点之间都可能相关，从而每个结点可有多个前趋和多个后继。图是所有数据结构中最一般的形式，树形结构、线性结构和集合都可看成形式受限的图。图结构有极强的表达能力，尤其是可以描述各种复杂的数据对象，因而应用也很广泛。

本章先介绍图的概念，然后介绍图的存储方法及有关图的一些算法和应用。

6.1 图的概念

图（Graph）是由若干个顶点与若干条边构成的结构，它的准确定义为：图 G 由两个集合 V 和 E 组成，记为 G=(V, E)，其中 V 是**顶点**（Vertex）的有穷非空集合，E 是**边**（Edge）的有穷集合，而边是 V 中顶点的偶对。通常，也将图 G 的顶点集和边集分别记为 V(G)和 E(G)。E(G)可以是空集，若 E(G)为空，则图 G 只有顶点而没有边。与其他数据结构类似，图中顶点是一些具体对象的抽象，边是对象间的关系。在具体应用中，顶点与边一般都有具体的含义，如顶点代表城市，边代表城市之间的道路等。

若图中的每条边都是有方向的，则称之为**有向图**（Digraph 或 Directed Graph）。有向图的边是顶点的有序偶对，一般用尖括号表示。例如，$<v_i, v_j>$表示一条有向边，v_i是边的始点（起点），v_j是边的终点。$<v_i, v_j>$和$<v_j, v_i>$是两条不同的边。有向边也称为**弧**（Arc），边的始点称为**弧尾**（Tail），终点称为**弧头**（Head）。例如，图 6.1 中 G_1 和 G_2 是有向图，其中对 G_1 而言，$V=\{v_1, v_2, v_3, v_4\}$，$E=\{<v_1, v_2>, <v_2, v_3>, <v_3, v_2>, <v_3, v_4>\}$。



（a）G_1　（b）G_2　（c）G_3　（d）G_4　（e）G_5

图 6.1　图的示例

若图的每条边都是没有方向的，则称之为**无向图**（Undigraph 或 Undirected Graph）。无向图的边是顶点的无序偶对，一般用圆括号表示。因此，(v_i, v_j)和(v_j, v_i)表示同一条边。例如，图 6.1 中 G_3～G_5 是无向图，其中对 G_3 而言，$V=\{v_1, v_2, v_3, v_4\}$，$E=\{(v_1, v_2), (v_1, v_3), (v_1, v_4)，(v_2, v_3), (v_2, v_4), (v_3, v_4)\}$。

在本章中，我们假定边的两个端点不相同（即不考虑顶点到其自身的边），每条边在图中也不重复出现。这样的图称作简单图。

若图中任意两个顶点之间均有边相连，则这种图称为**完全图**，确切地说，有 $C_n^2 = n(n-1)/2$ 条边的无向图称为**无向完全图**（Undirected Complete Graph），有 $P_n^2 = n(n-1)$条边的有向图称为**有向完全图**（Directed Complete Graph）。完全图具有最多的边数，一般情况下，图的顶点数 n 和边数 e 满足下述关系：对无向图，$0 \leqslant e \leqslant n(n-1)/2$；对有向图，$0 \leqslant e \leqslant n(n-1)$。例如图 6.1 中的 G_3 是具有 4 个顶点的无向完全图。

若图中的边数远远小于 n^2（即 $e<<n^2$），此类图称作**稀疏图**（Sparse Graph），若 e 接近于 n^2（准确地说，对无向图 e 接近于 $n(n-1)/2$，对有向图 e 接近于 $n(n-1)$），此类图称作**稠密图**（Dense Graph）。

对无向边(v_i, v_j)，称顶点 v_i 和 v_j 互为**邻接点**（Neighbors），或称 v_i 和 v_j 相**邻接**（Adjacent）；对有向边$<v_i, v_j>$，称顶点 v_i 邻接到 v_j，顶点 v_j 邻接于顶点 v_i，或者称 v_j 是 v_i 的邻接点。不论边是否有向，都称该边**关联**（Incident）于两个端点，或称该边与两个端点相关联。例如，在图 6.1 的 G_3 中，与顶点 v_3 相邻接的顶点是 v_1、v_2 和 v_4，而关联于顶点 v_3 的边是(v_1, v_3)、(v_2, v_3)和(v_3, v_4)；在 G_1 中，顶点 v_2 的邻接点是 v_3，关联于 v_2 的边是$<v_1, v_2>$、$<v_2, v_3>$和$<v_3, v_2>$，顶点 v_4 没有邻接点等。

顶点 v 的**度**（Degree）是指关联于该顶点的边的数目，记为 D(v)。对有向图，把以顶点 v 为终点的边的数目，称为 v 的**入度**（Indegree），记为 ID(v)；把以顶点 v 为始点的边的数目，称为 v 的**出度**（Outdegree），记为 OD(v)；显然顶点 v 的度等于其入度和出度之和，即 D(v)=ID(v)+OD(v)。有向边$<v_i, v_j>$也称为 v_i 的出边或 v_j 的入边。例如，对图 6.1 的 G_1，顶点 v_2 的入度为 2，出度为 1，度为 3。无论有向图还是无向图，所有顶点度数之和等于边数的 2 倍，即 $\sum_{i=1}^{n} D(v_i) = 2e$。

对图 G=(V, E)，从 V 中选出若干顶点组成子集 V′，从 E 中选出与 V′中顶点相关联的若干边组成子集 E′，则 G′=(V′, E′)也是一个图，称其为 G 的**子图**（Subgraph）。注意，E′中边的端点要在 V′中，否则(V′, E′)不是图，也就不可能是 G 的子图。图 6.2 给出了有向图 G_1 的若干子图，图 6.3 给出了无向图 G_3 的若干子图。

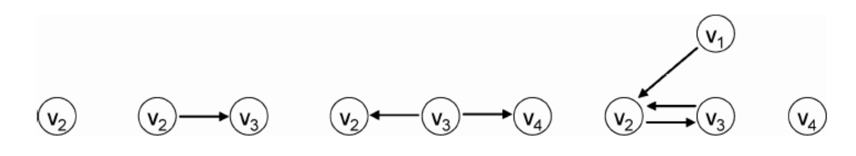


图 6.2　图 G_1 的若干子图

在无向图 G 中，若存在一个顶点序列 v_p，v_{i1}，v_{i2}，…，v_{ik}，v_q，使得$(v_p, v_{i1}), (v_{i1}, v_{i2}),\cdots, (v_{ik}, v_q)$均属于 E(G)，则称顶点 v_p 到 v_q 存在一条**路径**（Path），该路径也可简单地表示为

$(v_p, v_{i1}, v_{i2},\cdots, v_q)$。注意，路径中至少要有一条边。若 G 是有向图，则路径也是有向的，它由 E(G)中的有向边$<v_p, v_{i1}>, <v_{i1}, v_{i2}>,\cdots, <v_{ik}, v_q>$组成，也可表示为$<v_p, v_{i1}, v_{i2},\cdots, v_q>$。**路径长度**定义为路径上边的数目。若路径上除了 v_p 和 v_q 可以相同外，其余顶点均不相同，则称此路径为**简单路径**。起点和终点相同（$v_p=v_q$）的简单路径称为简单**回路**或简单**环**（Cycle）。例如，在图 G_3 中，顶点序列（v_1, v_2, v_3, v_4）是一条从顶点 v_1 到顶点 v_4 的长度为 3 的简单路径；顶点序列（v_1, v_2, v_4, v_1, v_3）是一条从顶点 v_1 到顶点 v_3 的长度为 4 的路径，但不是简单路径；顶点序列（v_1, v_2, v_3, v_1）是一个长度为 3 的简单环。在有向图 G_1 中，顶点序列（v_2, v_3, v_2）是一个长度为 2 的有向简单环。

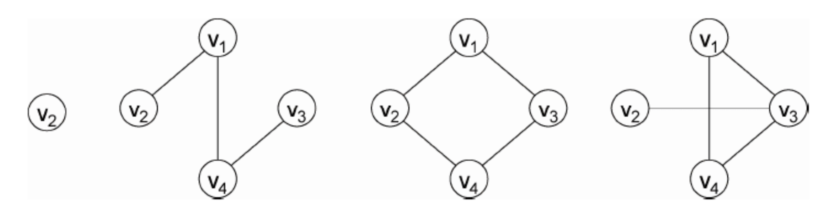


图 6.3　图 G_3 的若干子图

在一个有向图中，若存在一个顶点 v，从该顶点有路径可以到达图中其他所有顶点，则称此有向图为**有根图**，v 称作图的根。例如，图 G_1 就是有根图，其根为 v_1。

在无向图中，若从顶点 v_i 到顶点 v_j 有路径（当然从 v_j 到 v_i 也一定有路径），则称 v_j 和 v_i 是连通的。若图中任意两个不同的顶点都连通，则称该图为**连通图**（Connected Graph），如图 G_3 和图 G_4。显然，n 个顶点的连通图至少有 n−1 条边（对应某种树，如图 G_4）。

无向图的极大连通子图称为该图的**连通分量**（Connected Component）。显然，任何连通图的连通分量只有一个，即其自身，而非连通的无向图有多个连通分量。例如，图 G_5 是非连通图，它有 3 个连通分量，见图 6.4。

在有向图中，若任意两个不同的顶点 v_i 和 v_j，都存在从 v_i 到 v_j 以及从 v_j 到 v_i 的路径，则称该图为**强连通图**（Strongly Connected）。易知 n 个顶点的强连通图至少有 n 条边（对应某种有向回路，如图 G_2）或 0 条边（若图中只有 1 个顶点）。

有向图的极大强连通子图称为该图的**强连通分量**。显然，强连通图只有一个强连通分量，即其自身。非强连通的有向图可有多个强连通分量。例如图 G_1 不是强连通图，比如 v_2 到 v_1 就没有路径，但它有 3 个强连通分量，如图 6.5 所示。

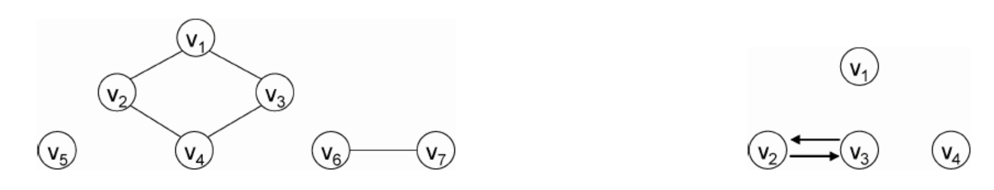


图 6.4　G_5 的三个连通分量

图 6.5　G_1 的三个强连通分量

在连通图中，若删去某顶点及其相关联的边后，可将图分割成两个或更多连通分量，则称该顶点为**关节点**。例如，图 6.6 中的顶点 v_3 就是关节点。没有关节点的连通图为**重连通图**，其中任何一对顶点之间至少存在两条路径，删去任一顶点及其相关联的边也不会破坏图的连通性。若通信、运输等网络是重连通的，则某个站点，或某条边出故障后，因剩

下部分仍连通，整个系统仍能正常运行。

若将图的每条边都赋上一个权，则称这种带权图为**网络**（Network）。通常权是具有某种意义的数，如表示两个顶点之间的距离、耗费等。图 6.7 就是一个网络例子。有时根据需要，图中顶点也可赋权。

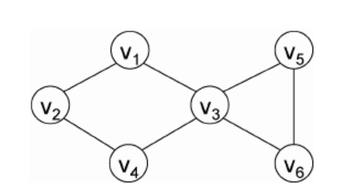


图 6.6　有关节点的流通图

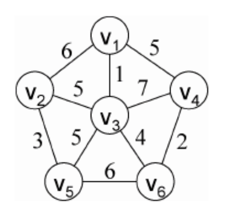


图 6.7　网络示例

在图中也可定义一些基本运算，如读顶点、求邻点、插入（顶点或边）、删除（顶点或边）等。为简单起见，本章不讨论这些基本运算，主要研究图的存储结构及一些常用运算的实现。

与树和其他结构类似，本章中对图的顶点编号从 1 开始，相应地，对图的存储和运算中涉及的一些数组也从下标 1 开始使用。这样对一维数组要牺牲 0 号单元，对二维数组要牺牲 0 列和 0 行单元，不过这些单元可作其他用途，如存放出度、入度信息等。这虽有一定的空间浪费，但使用比较方便。如果顶点编号从 0 开始、数组下标也从 0 开始使用，以下有关内容（主要是有关数组的下标表示）需略作修改。

6.2　图的存储

图中任一顶点都可能有多个前趋与多个后继，所以无法通过顶点的存储次序反映顶点间的逻辑关系，而必须显式地存储顶点之间的关系（即边）信息。图的存储方法较多，一般根据具体的应用和欲施加的操作进行选择。比如，最简单地可用一个数组存储顶点信息，用另一个数组存储边的信息，这适合对各边顺序处理的情况，其他则不方便（如边的插入、删除、查找等）。本节介绍两种常用的存储结构：邻接矩阵表示法和邻接表表示法。

6.2.1　邻接矩阵表示法

邻接矩阵（Adjacency Matrix）是表示顶点之间相邻关系的矩阵。设 G=(V, E)是具有 n 个顶点的图，则 G 的邻接矩阵是具有如下性质的 n 阶方阵：

$$A[i,j]=\begin{cases}1\text{：} & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{是}E(G)\text{中的边；}\\ 0\text{：} & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{不是}E(G)\text{中的边。}\end{cases}$$

矩阵的每个元素表示一条边，两个下标对应边的两个端点。例如，图 6.1 中的有向图 G_1 和图 6.3 无向图 G_3 的第 2 个子图的邻接矩阵分别为 A_1 和 A_2：

$$A_1=\begin{bmatrix}0&1&0&0\\0&0&1&0\\0&1&0&1\\0&0&0&0\end{bmatrix}\qquad A_2=\begin{bmatrix}0&1&0&1\\1&0&0&0\\0&0&0&1\\1&0&1&0\end{bmatrix}$$

显然，无向图的邻接矩阵是对称的，有向图就不一定；另外，邻接矩阵中 1 的个数对无向图为边数的两倍，对有向图则等于边数。

若 G 是网络，则邻接矩阵可定义为：

$$A[i,\ j]=\begin{cases}w_{ij}: & 若\ (v_i,v_j)\ 或 < v_i,v_j > \in E(G);\\ 0\ 或\ \infty: & 若\ (v_i,v_j)\ 或 < v_i,v_j > \notin E(G)。\end{cases}$$

其中，w_{ij} 表示边上的权值；∞表示一个计算机允许的、大于所有边上权值的数。对不存在的边，A[i, j]取 0 还是∞没有特别规定（但有关算法要与之一致，也可根据实际运算的需要或方便而定）。例如，图 6.7 中带权图的邻接矩阵就可有如下 3 种形式：

$$A_3=\begin{bmatrix}0&6&1&5&0&0\\6&0&5&0&3&0\\1&5&0&7&5&4\\5&0&7&0&0&2\\0&3&5&0&0&6\\0&0&4&2&6&0\end{bmatrix}\qquad A_4=\begin{bmatrix}\infty&6&1&5&\infty&\infty\\6&\infty&5&\infty&3&\infty\\1&5&\infty&7&5&4\\5&\infty&7&\infty&\infty&2\\\infty&3&5&\infty&\infty&6\\\infty&\infty&4&2&6&\infty\end{bmatrix}$$

$$A_5=\begin{bmatrix}0&6&1&5&\infty&\infty\\6&0&5&\infty&3&\infty\\1&5&0&7&5&4\\5&\infty&7&0&\infty&2\\\infty&3&5&\infty&0&6\\\infty&\infty&4&2&6&0\end{bmatrix}$$

本书采用的是第三种形式（见最小生成树、最短路径问题，它也比较自然）。

邻接矩阵一般用二维数组实现，但它只表示了顶点间的邻接关系，要完整地表示一个图，还要表示顶点本身的信息，这可另设一个顺序表来完成。

邻接矩阵表示的类型定义如下：

```
const int nmax=100;                    //顶点数的最大值，假设为 100
typedef struct {
  datatype data[nmax+1];               //顶点信息表，0 号单元不用
  mattype adjmat[nmax+1][nmax+1];      //邻接矩阵，0 行 0 列不用
  int n,e;                             //顶点数和边数
} mat_graph;
```

其中顶点信息表和邻接矩阵都从数组下标 1 开始使用。另外边数 e 是冗余的，因为可由邻接矩阵求出边数，增加该信息可方便一些涉及边数的运算。若图中各顶点的信息仅是一个编号，则不需要顶点信息数组。邻接矩阵的类型 mattype 可取为 char 以节省空间；若是网络，则取为权的类型，如 float。由于无向图或无向网络的邻接矩阵是对称的，还可采用压缩存储的方法，仅存储下三角（或上三角）矩阵中的元素，并且对角元也不必存储。

显然，邻接矩阵表示法的空间复杂度 $S(n)=O(n^2)$，与边数 e 无关，对稠密图比较有利；对稀疏图则会造成很大的浪费。不过，这种存储结构没有链指针之类的附加开销。

下面给出一个通过边的信息建立无向网络的算法。其中假设权的类型为 int，且每个顶点存放一个字符。在边（顶点对）的输入过程中，自动累计边数，若输入的顶点号为负则输入结束。

```
void creat_ga(mat_graph *g) {   //建立无向网络的邻接矩阵
  int i,j,n,e,w;
  cin>>n;                       //读入顶点数
  g->n=n;
  for(i=1;i<=n;i++) cin>>g->data[i];    //读入顶点信息，建立顶点表
  for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
      g->adjmat[i][j]=0;        //邻接矩阵初始化
  e=0;
  while(cin>>i>>j>>w,i>0) {     //读入边的端点号 i、j 和权 w，建立邻接矩阵
    e++;                        //累计边数
    g->adjmat[i][j]=w;
    g->adjmat[j][i]=w;
  }
  g->e=e;
}
```

该算法的执行时间是 $O(n+n^2+e)$，其中 $O(n^2)$的时间耗费在邻接矩阵的初始化操作上。由于边数 $e<n^2$，所以，算法的时间复杂度是 $O(n^2)$。

根据邻接矩阵的特点，下列运算规律是显然的：

（1）检查图中是否有边(v_i, v_j)或$<v_i, v_j>$，只要看对应的矩阵元素 A[i, j]是否非零即可。

（2）在图中增加或删除一条边，只需修改对应的矩阵元素。

（3）对有向图，某点 v_i 的出度等于第 i 行上非零元的个数，入度等于第 i 列上非零元的个数，度则等于第 i 行和第 i 列上非零元的个数之和。对无向图，第 i 行或第 i 列上非零元的个数就是 v_i 的度。

（4）找某点的邻接点，只需检查矩阵对应行上的非零元。

（5）若要求边数 e，须扫描整个矩阵，统计其中的非零元个数，所耗费的时间是 $O(n^2)$，与实际边数无关。

（6）若要增加或删除一个顶点，则要增加或删除矩阵中对应的行与列。

6.2.2　邻接表表示法

邻接表是图的一种链式存储结构，类似于树的孩子链表。在这种存储方法中，对图中的每个顶点 v_i 都建立一个单链表，其中记录所有邻接于该点的顶点。这个单链表就称为顶点 v_i 的**邻接表**（Adjacency List）。邻接表中的每个表结点有两个域，其一是邻接点域，用以存放与 v_i 相邻接的顶点的序号；其二是链域，用来将邻接表的所有表结点链在一起。如果要表示网络，则在每个表结点中增加一个权值域，存放相应边上的权。

每个顶点的邻接表都设置一个表头结点，它有两个域：一个是顶点域，用来存放顶点 v_i 的信息；另一个是指针域，用于存放指向 v_i 的邻接表中第一个表结点的头指针。

表结点和表头结点的组成形式见图 6.8，注意这里表头结点和表结点的类型不像一般链表那样完全相同。所有的表头顶点可以用单链表的形式链在一起，但一般为了便于管理和随机访问任一顶点的邻接表，将这些表头顶点顺序存储在一个向量中，称为**顶点表**。这样，图就可以由这个表头向量来表示。

显然，对于无向图而言，v_i的邻接表中每个表结点都对应于与v_i相关联的一条边；对于有向图来说，v_i的邻接表中每个表结点都对应于以v_i为始点的一条边。因此，我们又将无向图的邻接表称为**边表**，将有向图的邻接表称为**出边表**。例如，对于图 6.1 中的无向图 G_3，其邻接表表示如图 6.9 所示。

其中，顶点v_1的邻接表中 3 个表结点的顶点序号分别为 2、3 和 4，表示关联于v_1的边有 3 条(v_1, v_2)、(v_1, v_3)和(v_1, v_4)。而有向图G_1的邻接表表示如图 6.10（a）所示，其中顶点v_3的邻接表中有两个表结点，其顶点序号分别为 2 和 4，表示从v_3射出的两条边$<v_3, v_2>$和$<v_3, v_4>$。

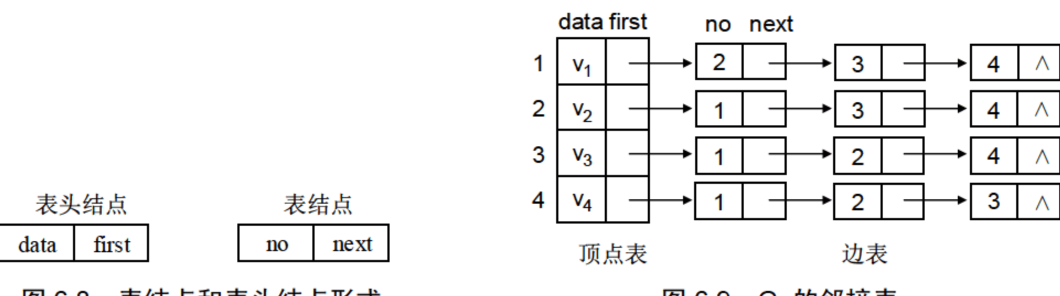
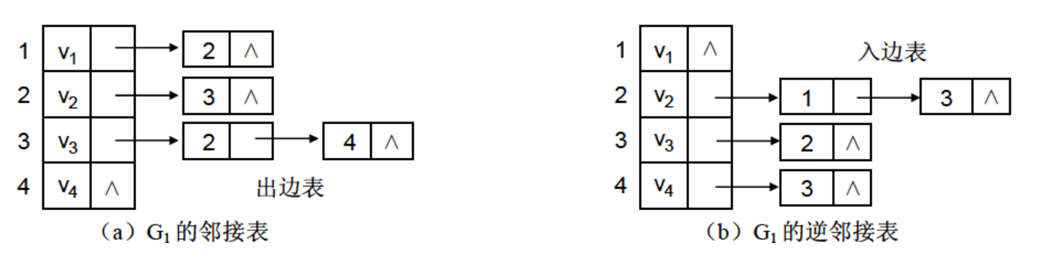


图 6.8 表结点和表头结点形式　　图 6.9 G_3的邻接表

有向图还有一种称为逆邻接表的表示法，该方法为图中每个顶点v_i建立一个**入边表**，入边表中的每个表结点均对应一条以v_i为终点（即射入v_i）的边。例如，G_1的逆邻接表如图 6.10（b）所示，其中v_2的入边表上有两个表结点，其顶点序号分别为 1 和 3，表示射入v_2的边有两条$<v_1, v_2>$和$<v_3, v_2>$。



（a）G_1的邻接表　　（b）G_1的逆邻接表

图 6.10 G_1的邻接表

显然，也可把有向图的邻接表和逆邻接表组合起来，即顶点表中每个结点存放两个指针，分别指向出边表和入边表，但这会增加顶点表的指针开销。也可把入边表链接到相应出边表的后面，但其中的顶点序号以负数形式存储（以便区分出边和入边），具体略。

邻接表表示的类型定义如下（其中边数 e 也是冗余的）：

```
const int nmax=100;              //顶点数的最大值，假设为100
typedef struct node * pointer;
struct node {                    //边表结点
  int no;                        //邻接点域
  pointer next;                  //链域
```

```
};
typedef struct {
  datatype data;                  //顶点信息
  pointer first;                  //边表头指针
} headtype;                       //顶点表结点类型
typedef struct {
  headtype adjlist[nmax+1];       //顶点表，0 号单元不用
  int n,e;                        //顶点数和边数
} lk_graph;
```

如果图中有 n 个顶点，e 条边，则邻接表需 n 个表头结点、e 个（对有向图）或 2e 个（对无向图）表结点。因此邻接表表示的空间复杂度为 S(n, e)=O(n+e)。显然，对稀疏图该存储方式的空间利用率较好，这与邻接矩阵的情况正好相反。对稠密图，考虑到邻接表中要附加较多链域，邻接矩阵表示法较好。与存储方式相对应，如果运算中要对所有的邻接点进行处理，则邻接矩阵表示和邻接表表示的时间复杂度一般分别为 $O(n^2)$和 O(n+e)。

下面给出建立无向图邻接表的一个方法。其中假设每个顶点存放的是一个字符。首先输入表头数组的顶点信息 data，并将每个表头的 first 域置为 NULL；然后读入顶点对(i, j)，生成两个边表结点，其 no 域分别置为 j 和 i，再将它们分别插入到第 i 个和第 j 个边表中。这里采用头插法。在顶点对输入过程中，自动累计边数，若输入的顶点号 i<0 则结束。

```
void creat_gl(lk_graph *g) {    //建立无向图的邻接表
  int i,j,n,e;
  pointer p;
  cin>>n;                         //读入顶点数
  g->n=n;
  for(i=1;i<=n;i++) {             //读入顶点信息，建立顶点表
    cin>>g->adjlist[i].data;
    g->adjlist[i].first=NULL;
  }
  e=0;
  while(cin>>i>>j,i>0) {          //读入边的顶点对序号，建立边表
    e++;                          //累计边数
    p=new node;                   //生成邻接点序号为 j 的表结点
    p->no=j;
    p->next=g->adjlist[i].first;
    g->adjlist[i].first=p;        //将新表结点插入到顶点 vi 的边表的头部
    p=new node;                   //生成邻接点序号为 i 的表结点
    p->no=i;
    p->next=g->adjlist[j].first;
    g->adjlist[j].first=p;        //将新表结点插入到顶点 vj 的边表的头部
  }
  g->e=e;
}
```

显然该算法的时间复杂度是 O(n+e)。建立有向图的邻接表与此类似，只是更加简单，每读入一个顶点对序号<i, j>时，仅需生成一个邻接点序号为 j 的表结点，将其插入到 v_i 的出边表头部即可。

对于网络，以上在邻接表类型定义时，边表结点中要增加一个数据域，用于存储边上的权；而在邻接表建立时，边的权值随边的顶点对序号一同输入并存入到相应的边表结点中，具体略。

值得注意的是，一个图的邻接矩阵表示是唯一的，但其邻接表表示不唯一。这是因为邻接表表示中，各边表中顶点的链接次序取决于建立邻接表的算法以及边的输入次序。以上述算法为例，它用头插法建立邻接表（边表），所以若图 G_3 输入的边依次为(1, 2)、(1, 3)和(1, 4)等，则图 6.9 中顶点 v_1 的边表中各结点的顺序为 4, 3, 2，而不是 2, 3, 4。

根据邻接表的特点，下列运算规律是显然的：

（1）对无向图，第 i 个边表中的结点个数就是顶点 v_i 的度。对有向图，第 i 个边表（即出边表）上的结点个数就是顶点 v_i 的出度；求入度较困难，须遍历所有的边表，累计其中邻接点域的值为 i 的表结点个数，即为顶点 i 的入度。对逆邻接表，情况正好相反，求顶点的入度容易，求顶点出度较难。

（2）判断图中是否有边(v_i, v_j)或$<v_i, v_j>$，须扫描第 i 个边表，最坏时间耗费为 O(n)。

（3）增加或删除一条边(v_i, v_j)，需要分别在 v_i 和 v_j 的邻接表中增加或删除邻接点域中顶点序号为 j 和 i 的表结点。若是有向边$<v_i, v_j>$，则只需要在 v_i 的邻接表中进行删除。

（4）增加一个顶点，需要在表头结点数组中增加一个元素；但删除一个顶点，不仅要在表头结点数组中删除相应的结点，还要在各边表中删除与之关联的所有边。

（5）求边数时需要扫描所有的边表，累计其中的表结点数，对有向图该数即为边数，对无向图该数为边数的 2 倍，所以时间耗费为 O(n+e)。

易见，如果将邻接矩阵看成稀疏矩阵，则邻接表（或逆邻接表）表示就是稀疏矩阵的行链表（或列链表）表示，其中每个边表对应于邻接矩阵的一行（或一列），边表中顶点的个数等于该行（或列）中非零元的个数。与稀疏矩阵的十字链表表示对应，图也有类似的表示，这就是邻接多重表表示（对有向图也称为十字链表表示），具体略。

6.3 图的遍历

与树类似，也可对图进行遍历，即从某个顶点出发，沿着某条搜索路径对图中所有顶点都作一次访问。若给定的图是连通图，则从图中任一顶点出发顺着边可以访问到该图的所有顶点；但若图中有回路，则可能在访问过程中又回到该顶点（但树中不存在回路），并可能导致死循环。为了避免顶点的重复访问，必须为每个顶点设立一个访问标志。为此一般有两种方法，一种是设置一个辅助数组 visited[1..n]，它的初值为 0，一旦访问了顶点 v_i，便将 visited[i]置为 1；另一种是在图的每个结点中增设一个访问标志域。后一种方法要修改图结点的存储结构，除了初始未访问标志可在建图的同时建立外，以后对图进行遍历后，若要清除访问标志则只有再“反遍历”一次，不方便。本节以下采用的是辅助数组的方法。

与线性表和树的遍历不同的是，图的遍历可以从任一点开始，即遍历序列的第一个点可以是图中任一点。作为对比，线性表遍历的第一个点一般是线性表的开始结点，树遍历的第一个点一般是根（如先根遍历）或左子树中最左下的点（如后根遍历）。

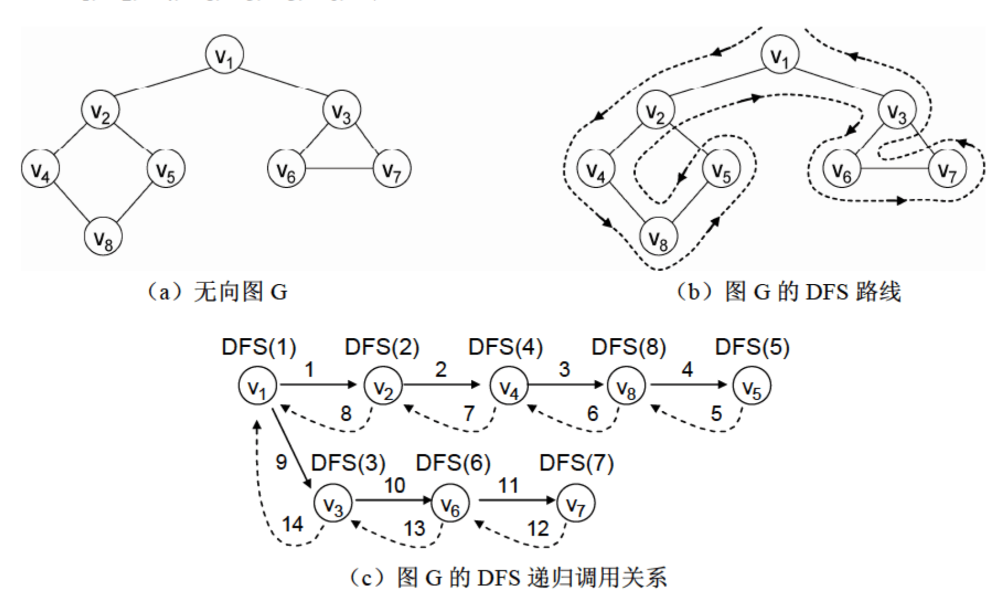
根据搜索路径的方向不同，图有两种常用的遍历方法：深度优先搜索遍历和广度优先搜索遍历，分别对应于树的先根遍历与层序遍历。

6.3.1 连通图的深度优先搜索遍历

深度优先搜索（Depth First Search，DFS）类似于树的先根遍历，可认为是树的先根遍历的推广。它的基本思想是：在图中任选一顶点 v_i 为初始出发点，首先访问出发点 v_i（并标记为已访问），然后依次搜索 v_i 的每一个邻接点 v_j，若 v_j 未访问过，则以 v_j 为新的出发点继续进行深度优先搜索；依此类推，直到访问完所有和 v_i 有路径的顶点。这一过程也可简单地描述为：访问出发点，然后递归地访问邻接于此点的所有顶点。

这种搜索方法的特点是尽可能先对纵深方向搜索，故称为深度优先搜索。其直观表现是，先沿某一分支搜索到尽头，然后回溯，再沿另一分支搜索。在深度优先搜索过程中，将得到一个按访问先后次序排列的顶点序列，称为该图的深度优先搜索遍历序列，简称 DFS 序列。

以图 6.11（a）的无向图 G 为例，设初始出发点是 v_1，首先访问 v_1，并将其标记为已访问。v_1 有两个邻接点 v_2 和 v_3，它们都未访问过，任选一个作为新的出发点，这里假设选 v_2。于是访问 v_2 并将其标记为已访问，再找 v_2 的邻接点，有 v_1、v_4 和 v_5 共 3 个，但 v_1 已访问过，故在 v_4 和 v_5 中选一个，假设取 v_4 作为新的出发点。类似，继续依次访问 v_8 和 v_5。访问 v_5 后，由于它的两个邻接点 v_2 和 v_8 均已访问过，于是搜索回退。先回退到 v_8，但 v_8 的两个邻接点也已访问过，继续回退到 v_4。类似又由 v_4 回退到 v_2，再回退到 v_1。这时 v_1 有未访问过的邻接点 v_3，于是以它为新的出发点，继续搜索，依次访问 v_3、v_6、v_7，最后又由 v_7 回退到 v_6、v_3、v_1。此时，v_1 的所有邻接点都已访问，搜索结束。这个过程得到的 DFS 序列为 $v_1, v_2, v_4, v_8, v_5, v_3, v_6, v_7$。



（a）无向图 G （b）图 G 的 DFS 路线

（c）图 G 的 DFS 递归调用关系

图 6.11 深度优先搜索示例

上述执行过程可用图 6.11（b）、（c）表示，其中子图（b）的包络线表示搜索路线，每个点都经过了至少一次：第一次经过某点 v 时表示访问该点，以后再经过该点时表示回退。将搜索路线中所有第一次经过的点列表即得图的 DFS 序列。子图（c）表示递归调用关系，

实线表示递归，虚线表示回退，线上的数字表示调用和返回的次序。

从上述过程可知，图的 DFS 序列不一定唯一，首先，如果初始出发点不同，结果肯定不同；其次，如果搜索中某个点有多个未访问的邻接点时，则选取不同的邻接点结果也不同。如何找邻接点以及选取哪个邻接点，涉及图的具体存储结构和算法。所以，DFS 序列与算法、图的存储结构以及初始出发点有关。在初始点和存储结构一定时，如果在选邻接点的过程中加些约束，如有多个候选点时取顶点序号小的一个，则结果就唯一了。

深度优先搜索法是递归定义的，可很容易地写出其递归算法：

```
void DFS(graph g,int v) {
  访问 v;visited[v]=1;
  找出 g 中 v 的一个邻接点 w;
  while(w 存在) {
    if(w 未访问过) DFS(g,w);
    w=g 中 v 的下一个邻接点;
  }
}
```

下面分别以邻接矩阵和邻接表作为图的存储结构给出具体算法，算法中假设 visited 为全程量，且各分量已初始化为 0，算法如下。

```
void dfs(mat_graph *g,int v) { //邻接矩阵上 DFS 遍历
  int j;
  cout<<v<<" ";visited[v]=1;    //访问出发点，假设为输出顶点序号
  for(j=1;j<=n;j++)             //依次搜索 v 的邻接点 j，若未访问，则从 j 出发递归
    if(g->adjmat[v][j]==1 && !visited[j]) dfs(g,j);
}
void dfsL(lk_graph *g,int v) { //邻接表上 DFS 遍历
  pointer p;
  cout<<v<<" ";visited[v]=1;    //访问出发点，假设为输出顶点序号
  p=g->adjlist[v].first;
  while(p!=NULL) {              //依次搜索 v 的邻接点，若未访问，则递归
    if(!visited[p->no]) dfsL(g,p->no);
    p=p->next;
  }
}
```

对算法 dfs，从出发点 v 搜索时，是在邻接矩阵的第 v 行从左到右寻找下一个未访问过的邻接点，若这种邻接点有多个，则选中的是序号小的那一个。由于图的邻接矩阵表示是唯一的，故对于指定的初始出发点，dfs 算法对同一个图得到的 DFS 序列是唯一的。

对算法 dfsL，找 v 的邻接点时，是在该点的邻接表（边表）中从前向后寻找下一个未访问过的邻接点，若这种邻接点有多个，则选中的是先找到的那一个。于是对一个具体的邻接表表示来说，从初始出发点得到的 DFS 序列也是唯一的。但图的邻接表表示并不唯一，它取决于边表中结点的链接次序，从而对指定的初始出发点，由于邻接表的不同，算法 dfsL 对同一个图得到的 DFS 序列就不一定唯一了。

对于有 n 个顶点 e 条边的连通图，算法 dfs 和 dfsL 均递归调用 n 次。在每次递归调用时，除访问顶点及做标记外，主要时间耗费在从该顶点出发搜索它的所有邻接点。用邻接矩阵表示图时，搜索一个顶点的所有邻接点需检查矩阵相应行中所有的 n 个顶点，要花费 O(n)的时间，故从 n 个顶点出发搜索所需的时间是 $O(n^2)$，即 dfs 算法的时间复杂度是 $O(n^2)$。

用邻接表表示图时，搜索 n 个顶点的所有邻接点需将各边表结点扫描一遍，而边表结点的总个数为 2e（无向图）或 e（有向图），故算法 dfsL 的时间复杂度为 O(n+e)。算法 dfs 和 dfsL 所用的辅助空间是标志数组和实现递归所用的栈，它们的空间复杂度为 O(n)。

顺便指出，DFS 是算法策略“**回溯法**”的基础，它将问题的候选解集（解空间）组织成某种树或图，用 DFS 搜索该空间，并采用一定的规则避免移动到无可行解的子空间，以提高搜索效率。搜索的同时产生可行解，空间复杂度为 O(最长路径长度)。

6.3.2　连通图的广度优先搜索遍历

广度优先搜索（Breadth/Width First Search，BFS）类似于树的层序遍历，可认为是树的层序遍历的推广。它的基本思想是：在图 G 中任选一顶点 v_i 为初始出发点，首先访问出发点 v_i（并标记为已访问），接着依次访问 v_i 的邻接点 w_1, w_2, …, w_t，然后，依次访问与 w_1, w_2,…, w_t 邻接的所有未访问过的顶点，依此类推，直到访问完所有和 v_i 有路径的顶点。

上述搜索方法的特点是尽可能先横向搜索，故称为广度优先搜索。其直观表现是，从出发点开始，“一层一层”地进行搜索。从这点上看，广度优先遍历相当于对图进行了分层（与出发点不连通的结点层数为∞）。和定义图的 DFS 序列类似，对图进行广度优先搜索遍历得到的顶点序列，称为该图的广度优先搜索遍历序列，简称 BFS 序列。

以图 6.11（a）的无向图 G 为例，设出发点为 v_1，则 BFS 的执行过程是：首先访问出发点 v_1；它有两个未访问的邻接点 v_2 和 v_3，设先访问 v_2 再访问 v_3；然后再先后访问 v_2 的未访问过的邻接点 v_4 和 v_5，接着先后访问 v_3 的未访问过的邻接点 v_6 和 v_7；再下来是访问 v_4 的未访问过的邻接点 v_8；以后再依次访问 v_5、v_6、v_7 和 v_8 的未访问过的邻接点，但都没有，搜索结束。于是得到的 BFS 序列是 v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8。

一个图的 BFS 序列也不是唯一的，它与算法、图的存储结构及初始出发点有关，比如，在搜索中若某个点有多个未访问的邻接点时，对它们按不同的顺序访问，结果就不相同。但不难看到，从给定出发点开始的“层次关系”是相同的。

在广度优先搜索中，先访问的顶点其邻接点也先访问，即有“先进先出”的特点，所以 BFS 遍历算法要借助队列来实现。非形式算法如下：

```
void BFS(graph g,int v) {
  初始化队列;
  访问 v;visited[v]=1;v 入队;
  while(队不空) {
    v 出队;
    找 v 的第一个邻接点 w;
    while(w 存在) {
      if(w 未访问过) {访问 w;visited[w]=1;w 入队;}
      求 v 的下一个邻接点 w;
    }
  }
}
```

该算法在顶点访问后入队（队列中保存已访问过的顶点），若改为出队后访问（队列中保存将要访问的顶点，就像第 5 章树的层序遍历那样），则出队后要先检测访问标志后再

访问（虽然入队时已检测过），这是因为有些顶点可能多次入队而以后也多次出队“访问”（这些点是一些顶点的共同邻点，但树中不同结点不会有共同的邻点——孩子）。

假设对顶点的访问是输出顶点序号，并假设 visited 为全程量，且各分量已初始化为 0，分别以邻接矩阵和邻接表作为图的存储结构，则广度优先搜索算法如下：

```
void bfs(mat_graph *g,int v) { //邻接矩阵上 BFS 遍历
  int j;
  sqqueue Q;                    //假设采用顺序队列
  init_sqqueue(&Q);
  cout<<v<<" ";visited[v]=1;    //访问出发点，假设为输出顶点序号
  en_sqqueue(&Q,v);
  while(!empty_sqqueue(&Q)) {
    de_sqqueue(&Q,&v);
    for(j=1;j<=g->n;j++)
      if(g->adjmat[v][j]==1 && !visited[j])
        {cout<<j<<" ";visited[j]=1;en_sqqueue(&Q,j);}
  }
}
void bfsL(lk_graph *g,int v) { //邻接表上 BFS 遍历
  sqqueue Q;                    //假设采用顺序队列
  pointer p;
  init_sqqueue(&Q);
  cout<<v<<" ";visited[v]=1;    //访问出发点，假设为输出顶点序号
  en_sqqueue(&Q,v);
  while(!empty_sqqueue(&Q)) {
    de_sqqueue(&Q,&v);
    p=g->adjlist[v].first;
    while(p!=NULL) {
      if(!visited[p->no])
       {cout<<p->no<<" ";visited[p->no]=1;en_sqqueue(&Q,p->no);}
      p=p->next;
    }
  }
}
```

与深度优先遍历类似，对同一个图和指定的初始出发点，由于邻接矩阵唯一，bfs 算法得到的 BFS 序列唯一；由于邻接表不唯一，bfsL 算法得到的 BFS 序列不唯一（但对给定的邻接表其结果是唯一的）。

对于具有 n 个顶点和 e 条边的连通图，因为每个顶点均出队一次，所以算法 bfs 和 bfsL 的外循环次数为 n。算法 bfs 的内循环是 n 次，故算法 bfs 的时间复杂度为 $O(n^2)$。算法 bfsL 的内循环次数取决于各顶点的边表结点个数，但内循环执行的总次数是边表结点的总个数 2e（无向图）或 e（有向图），故算法 bfsL 的时间复杂度是 O(n+e)。算法 bfs 和 bfsL 所用的辅助空间是队列和标志数组，故它们的空间复杂度为 O(n)。

这些结果与深度优先遍历时相同（但 BFS 所需队列空间较大时，DFS 所需栈空间较小，反之亦然）。

6.3.3 非连通图的遍历

对一个无向图，若它是非连通的，则从图中任意一个顶点出发进行 DFS 或 BFS 都不能访问到图中所有顶点，而只能访问到初始出发点所在的连通分量中的所有顶点。但如果

从每个连通分量中都选一个顶点作为出发点进行搜索，则可访问到整个非连通图的所有顶点。因此非连通图的遍历必须多次调用 DFS 或 BFS 算法。

以邻接矩阵为例，非连通图 DFS 遍历算法如下：

```
void traver(mat_graph *g) {          //邻接矩阵上非连通图的 DFS 遍历
  int i,count;
  for(i=1;i<=n;i++) visited[i]=0;   //初始化标志数组
  count=0;
  for(i=1;i<=n;i++) {
    if(!visited[i]) {
      count++;
      cout<<"第"<<count<<"个连通分量: ";
      dfs(g,i);
      cout<<"\n";
    }
  }
}
```

若算法 traver 只调用了一次 DFS（或 BFS），则表示图是连通的；否则，调用了几次就表示图中有几个连通分量。例如，对图 6.1（e）所示的非连通图 G_5，执行算法 traver 时，分别调用了 DFS(1)、DFS(5)和 DFS(6)。输出的 3 个连通分量是：

```
第 1 个连通分量: 1  2  4  3
第 2 个连通分量: 5
第 3 个连通分量: 6  7
```

不论图是否连通，每个顶点都要调用 DFS 一次，所以算法 traver 的时间复杂度是 $O(n^2)$。

显然，将上述算法中的 DFS 调用换成 BFS 调用，则得到 BFS 遍历算法；将邻接矩阵换成邻接表则得到邻接表上的遍历算法，但这时算法的时间复杂度为 O(n+e)。

以上讨论的各种遍历算法是以无向图为例的，但算法本身对有向图也是适用的，只是要注意有向图的路径是有方向的，比如，若从某个顶点出发按出度遍历和按入度遍历得到的顶点集相同，则该顶点集即对应一个强连通分量。

6.4　生成树

在图论中，常将树定义为无回路的连通图。例如，图 6.12 就是两个无回路的连通图。乍一看它们似乎不是树，但只要选定某个顶点做根，以树根为起点对每条边定向，就能将它们变为通常的树。由于没有确定的根，这种图又称为**自由树**（Free Tree）或者**树图**[①]。

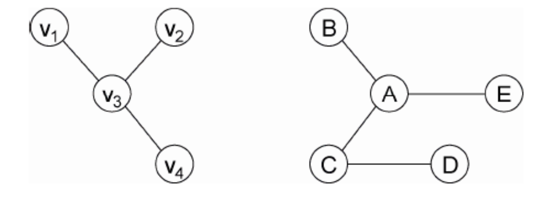


图 6.12　两个无回路的连通图

① 相应地，可把之前讨论的有确定根的树称为**有根树**（Rooted Tree）。

连通图 G 的一个子图如果是一棵包含 G 的所有顶点的树，则该子图称为 G 的**生成树**（Spanning Tree）。由于 n 个顶点的连通图至少有 n−1 条边，而所有包含 n−1 条边及 n 个顶点的连通图都是无回路的树，所以生成树是连通图的极小连通子图。所谓极小是指边数最少，若在生成树中去掉任何一条边，都会使之变为非连通图；若在生成树上任意添加一条边，就必定出现回路。注意，这里的生成树是从连通图的观点出发、针对无向图而言的，下面还会给出生成树的另一种定义，对有向图和无向图都适用。

对给定的连通图，如何求其生成树呢？

设图 G=(V, E)，在深度优先搜索或广度优先搜索中，从一个已访问的顶点 v_i 搜索到一个未访问的邻接点 v_j，必定要经过 G 中的一条边(v_i, v_j)。如果 G 是一个具有 n 个顶点的连通图，则从 G 的任一顶点出发，可将 G 中的所有 n 个顶点都访问到。这样，除初始出发点外，对其余 n−1 个顶点的访问一共经过 G 中的 n−1 条边，这些边正好将 G 的 n 个顶点连接成一个极小连通子图，从而得到 G 的一棵生成树。

具体地说，在算法 dfs（或 dfsL）中，当 dfs(i)要调用 dfs(j)时，v_i 是已访问过的顶点，v_j 是邻接于 v_i 的、未曾访问过且正待访问的顶点。因而在 dfs 算法的 if 语句中，在递归调用 dfs(j)前插入适当的语句，将边(v_i, v_j)打印或保存起来，就可得到求生成树的算法。

类似地，在算法 bfs（或 bfsL）中，若当前出队的元素是 v_i，待入队的元素是 v_j，则 v_i 是已访问过的顶点，v_j 是待访问而未曾访问过的、邻接于 v_i 的顶点。因而在 bfs 算法的 if 语句中插入适当语句，也可得到求生成树的算法。

由深度优先搜索得到的生成树称为深度优先生成树，简称为 **DFS 生成树**；由广度优先搜索得到的生成树称为广度优先生成树，简称为 **BFS 生成树**。一般可将初始出发点看成生成树的根，这时 BFS 生成树的高度一般比 DFS 生成树小。例如，从图 6.11（a）的顶点 v_1 出发得到的 DFS 生成树和 BFS 生成树，如图 6.13 所示。

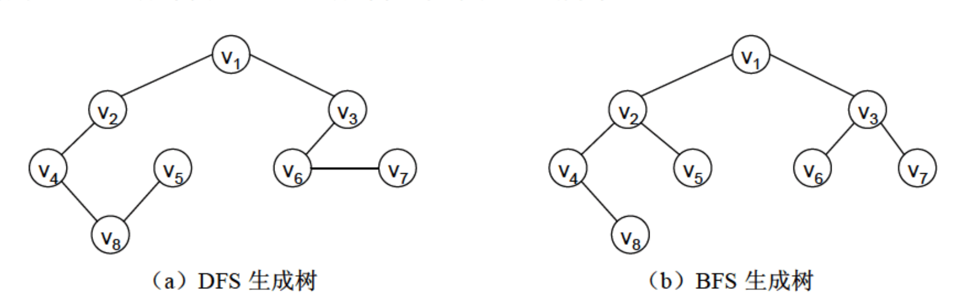
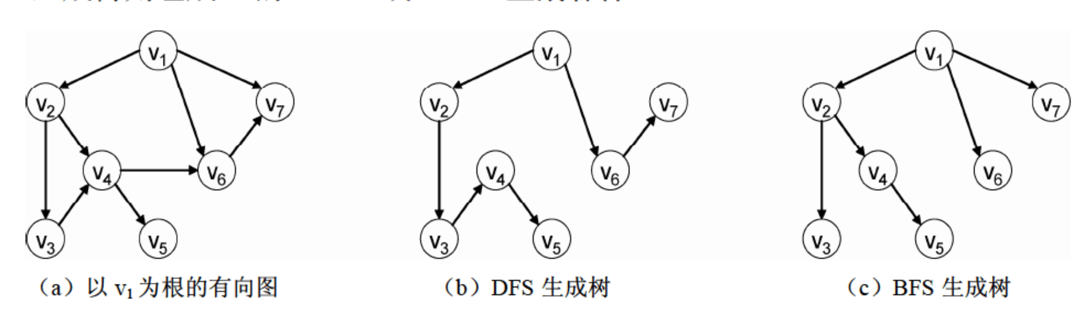


图 6.13　DFS 和 BFS 生成树

由于从图的遍历可求得生成树，因此也可以将生成树定义为：若从图的某顶点出发，可以系统地访问到图中所有顶点，则遍历时经过的边和图的所有顶点所构成的子图，称作该图的生成树。这个定义不仅适用于无向图，对有向图也同样适用。

显然，若 G 是强连通的有向图，则从其中任一顶点 v 出发，都可以访问遍 G 中的所有顶点，从而得到以 v 为根的生成树。若 G 是有根的有向图，设根为 v，则从 v 出发也可以完成对 G 的遍历，因而也能得到 G 的以 v 为根的生成树。例如，图 6.14（a）是以 v_1 为根的有向图，它的 DFS 生成树和 BFS 生成树分别如图 6.14（b）和图 6.14（c）所示。有向图的生成树也是有方向的，它们属于**有向树**。

若 G 是非连通的无向图，则要若干次从外部调用 DFS（或 BFS）算法，才能完成对 G 的遍历。每一次外部调用，只能访问到 G 的一个连通分量的顶点集，这些顶点和遍历时所经过的边构成该连通分量的一棵 DFS（或 BFS）生成树。G 的各个连通分量的 DFS（或 BFS）生成树则组成 G 的 DFS（或 BFS）**生成森林**。



（a）以 v_1 为根的有向图　（b）DFS 生成树　（c）BFS 生成树

图 6.14　有向图及其生成树

类似地，若 G 是非强连通的有向图，且初始出发点又不是有向图的根，则遍历时一般也只能得到该有向图的生成森林。

6.5 最小生成树

图的生成树不唯一，从不同的顶点出发进行遍历，可以得到不同的生成树。如果图 G 是一个连通网络，由于边是带权的，则其生成树的各边也是带权的。我们把生成树中各边权值的总和称为生成树的权，并把权最小的生成树称为图 G 的**最小生成树**（Minimun Spaning Tree，MST）。

生成树和最小生成树有许多重要的应用。设图 G 的顶点表示城市，边表示连接两个城市之间的通信线路。n 个城市之间最多可设立的线路有 n(n−1)/2 条，把 n 个城市连接起来至少要有 n−1 条线路，则图 G 的生成树表示了建立通信网络的可行方案。如果给图中的边都赋予权，表示两个城市之间通信线路的长度或造价，那么，如何选择 n−1 条线路，使得所建通信网络的总长度最短或总代价最小？这就需要构造该图的一棵最小生成树。

最小生成树的建立，一般是个逐步进行的过程。这个过程可能有多种方法，其中有一种十分有效的方法，其基本原则是使每步满足以下条件：

（1）当前生成的图是连通的。

（2）当前生成的图不含回路。

（3）当前生成的图是最小生成树的一部分。

这是一种**贪心算法**（Gready），即“步步为营”，每一步所做的事情，都是最终结果所必要的，不做任何“多余”的事情，所以这类方法的效率一般较高。按上述原则生成的图，一定是最小生成树，该原则也可作为证明此类算法正确性的一个公理。

以下我们只讨论无向图的最小生成树问题。构造最小生成树的算法中大多都利用了最小生成树的下述性质。

MST 性质：设 G=(V, E)是一个连通网络，U 是顶点集 V 的一个真子集。若(u, v)是 G 中所有的一个端点在 U 里（即 $u \in U$）、另一个端点不在 U 里（即 $v \in V-U$）的边中的权值最小的一条边，则一定存在 G 的一棵最小生成树包含此边(u, v)。

该性质可用反证法证明：假设 G 的任何最小生成树中都不含边(u, v)。设 T 是 G 的一棵最小生成树，但不包含边(u, v)。由于 T 是树，且是连通的，因此有一条从 u 到 v 的路径，且该路径上必有一条连接两个顶点集 U 和 V−U 的边(u′, v′)，其中 $u' \in U$，$v' \in V-U$，否则 u 和 v 不连通。若把边(u, v)加入到树 T，则得到一个含有边(u, v)的回路，见图 6.15。删去边(u′, v′)，上述回路即被消除，由此得到另一棵生成树 T′。T′和 T 的区别仅在于用边(u, v)取代了 T 中的边(u′, v′)。因为(u, v)的权≤ (u′, v′)的权，故 T′的权≤T 的权，因此 T′也是 G 的最小生成树，但它包含了边(u, v)，与假设矛盾！

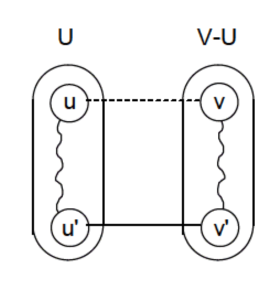


图 6.15 包含边(u,v)的回路

本节将介绍两个著名的算法——**普里姆**（Prim，1957 年发表）算法和**克鲁斯卡尔**（Kruskal，1956 年发表）算法，它们都属于贪心算法。

6.5.1 Prim 算法

设 G=(V, E)为网络，顶点集为 V={1, 2, …, n}；最小生成树为 T=(U, TE)，其中 U 是 T 的顶点集，TE 是 T 的边集；并且将 G 中边上的权看做边的长度。

Prim 算法的基本思想是：首先从 V 中任取一个顶点 u_0，将生成树 T 置为仅有一个结点 u_0 的树，即置 U={u_0}，TE=∅。然后只要 U 是 V 的真子集，就在所有一个端点在 U 中、另一个端点在 U−V 中的边中，找一条最短（即权最小）的边(u, v)，并把该边(u, v)加入到边集 TE，顶点 v 加入到顶点集 U。如此进行下去，每次往生成树里加入一个顶点和一条边，直到把所有顶点都包括进生成树 T 为止。可见，Prim 算法是个逐步扩大 U 和 TE 的过程，最终必有 U=V，TE 中有 n−1 条边。

关于 Prim 算法的正确性，只要说明该算法满足上节提到的三项原则即可，证明如下：

（1）每次构成的部分树是连通的。最初的部分树只有一个点，显然是连通的。以后每次新加入的点对应一条新加入的边，而该边有一个端点属于原连通的部分树，故新的部分树也是连通的。

（2）每次构成的部分树是无回路的。因为每次加入的边的两个端点原属于两个不同集合 U 和 V−U，不可能构成回路。只有两个端点同在 U 中才有可能构成回路。

（3）每次构成的部分树是最小生成树的一部分。MST 性质保证了每次加入的边和顶点是最小生成树的边和顶点。

显然，Prim 算法的关键是如何找到连接 U 和 V−U 的最短边来扩充生成树。设当前生成的 T 中有 k 个顶点，则连接 U 和 V−U 的边有 k(n−k)条（若两点间没有边，则设想有一条长度为∞的虚拟边）。若每次都直接从如此多的边中选取一个最短边，则运算量大，效率不高，因为找当前最短边要进行一系列比较，而这些比较很多在以前找最短边时已进行过，但没有把结果保留下来，导致很多重复的比较。

为此，可构造一个较小的候选边集，且保证最短边属于该候选边集。注意到对于 V−U

中的每个点，它到 U 的各边中，都有一条自己的最短边（为叙述方便，以下简称为该点的最短边），这些边共有 n−k 条，其中的最短边也就是原 k(n−k)条边中的最短边，所以我们把这些边作为候选边集。于是，扩充 T 就是从候选边集中选出最短边(u, v)，将它连同顶点 v 加入到 T 中。此时，V−U 集中原来与 v 相连的边变成了连接新的 V−U 和 U 的候选边。这时，可对候选边集进行如下调整：若原 V−U 集中，点 i 的最短边大于新的候选边(v, i)，则以(v, i)作为 i 的新最短边；否则 i 的最短边不变。

Prim 算法的非形式描述如下：

```
置 T 为任意一个顶点;
求初始候选边集;
while(T 中结点数<n) {
  从候选边集中选取最短边(u,v);
  将(u,v)及顶点 v，扩充到 T 中;
  调整候选边集;
}
```

图 6.16 给出了一个例子说明按上述算法构造最小生成树的过程：

开始时，取顶点集 U={1}，V−U={2, 3, 4, 5, 6}，初始的候选边集是 V−U 的 5 个点与 U 的顶点 1 所关联的最短边，如图 6.16（b）所示。其中，顶点 5 和 6 与顶点 1 没有关联边，取它们的最短边为无穷大。在这 5 条最短边中，边(1, 3)的长度最短，因此，将该边扩充到 T 中，U={1, 3}。

因为顶点 3 加入 U，候选边集调整如下：顶点 2 的原最短边(1, 2)的长度为 6，而新候选边(3, 2)的长度为 5，比前者小，因此用(3, 2)取代(1, 2)作为顶点 2 的最短边；同理，用新候选边(3, 5)和(3, 6)分别取代顶点 5 和 6 的原最短边(1, 5)和(1, 6)；顶点 4 的原最短边(1, 4)长度为 5，小于新候选边(3, 4)的长度 7，所以顶点 4 的最短边不变。调整后的候选边集如图 6.16（c）的 4 条虚线所示。选择其中最短的一条边(3, 6)扩充到 T 中，U={1, 3, 6}。

如此进行下去，最终得到的生成树 T 即为所求的最小生成树，如图 6.16（g）所示。

若候选边集中的最短边不止一条时，可任选其中的一条扩充到 T 中，因此，连通网络的最小生成树不一定唯一，但它们的权是相等的。例如在图 6.16（e）中若选取的最短边是(3, 5)而非(3, 2)时，则得到另一棵最小生成树，如图 6.17 所示。

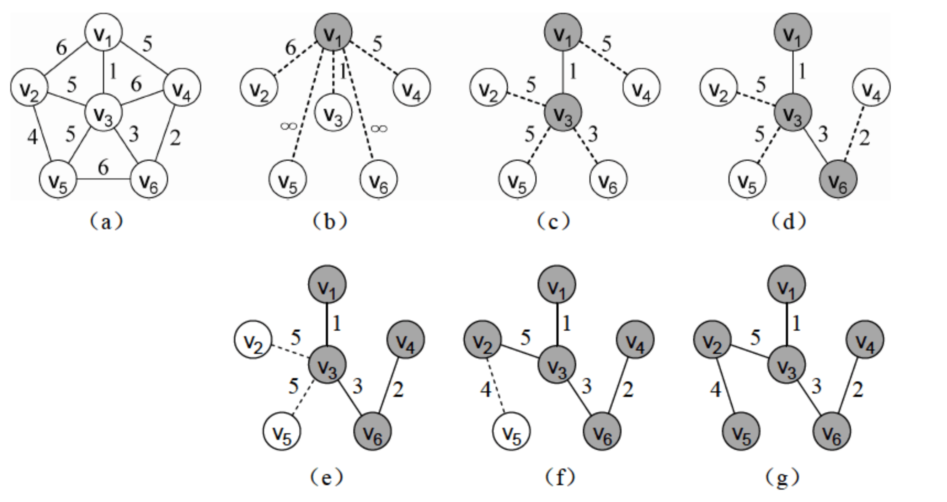


图 6.16　Prim 算法构造最小生成树的过程

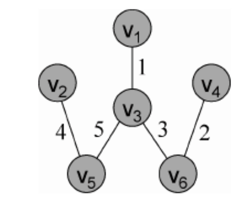


图 6.17　图 6.16（a）的另一棵 MST

从图 6.16 可以看到，在 Prim 算法的各个中间时刻，已生成的树与候选边集中的所有边一起可构成一棵生成树，但不一定是最小生成树。

上述过程也可用表 6.1 表示，其中方括号内的数值为顶点的候选边长度，阴影部分表示已选出的顶点（及边），但不如图形表示简洁。

表 6.1 Prim 算法求最小生成树的过程

顶点	1	2	3	4	5	6	最短边 (u, v)	U	V−U	图例 (图 6. 16)
候选边终点[边长]		1[6]	1[1]	1[5]	1[∞]	1[∞]	(1, 3)	{1}	{2, 3, 4, 5, 6}	(b)
候选边终点[边长]		3[5]		1[5]	3[5]	3[3]	(3, 6)	{1, 3}	{2, 4, 5, 6}	(c)
候选边终点[边长]		3[5]		6[2]	3[5]		(6, 4)	{1, 3, 6}	{2, 4, 5}	(d)
候选边终点[边长]		3[5]			3[5]		(3, 2)	{1, 3, 6, 4}	{2, 5}	(e)
候选边终点[边长]					2[4]		(2, 5)	{1, 3, 6, 4, 2}	{5}	(f)
候选边终点[边长]								{1, 3, 6, 4, 2, 5}	∅	(g)

下面讨论 Prim 算法的实现。设连通网络用邻接矩阵表示，对不存在的边，相应的矩阵元素为∞（实取为计算机允许的最大数，也可取为大于所有边权的一个数）。

边的存储结构如下：

```
struct {
  int end;            //最短边的终点(起点是候选点自己)
  int len;            //边长，假设权为整数
} minedge[nmax+1];   //候选边集，nmax 为顶点数最大值。数组下标从 1 开始使用
```

其中，候选边按候选点的序号排列，边的序号就是候选点号，也就是候选边的起点，故候选边信息中没有存放起点信息。假设候选边集数组为全局量，则 Prim 算法如下：

```
void prim(mat_graph *g,int u) {      //从顶点 u 出发构造最小生成树
  int v,k,j,min;
  for(v=1;v<=g->n;v++) {             //初始化，构造初始候选边集
    minedge[v].end=u;
    minedge[v].len=g->adjmat[v][u];
  }
  minedge[u].len=0;                  //顶点 u 并入 U 集，边权置 0 避免重复选取
  for(k=1;k<g->n;k++) {              //依次找 n-1 条最短边
    min=INT_MAX;                     //INT_MAX 为整数最大值，表示∞
    v=0;                             //v 用于记录最短边号(此处可不必设初值)
    for(j=1;j<=g->n;j++)             //在候选边集中找最短边
      if(minedge[j].len>0 && minedge[j].len<min) {
        min=minedge[j].len;
        v=j;
      }
    if(min=INT_MAX) {cout<<"图不连通，无生成树！";exit(0);}
    cout<<v<<" "<<minedge[v].end<<endl;      //输出生成树的边
    minedge[v].len=-minedge[v].len; //顶点 v 并入 U 集，边权置负避免重复选取
    for(j=1;j<=g->n;j++)             //调整候选边集
      if(g->adjmat[j][v]<minedge[j].len) {
        minedge[j].len=g->adjmat[j][v];
        minedge[j].end=v;
```

```
        }
    }
}
```

显然，该算法的时间复杂度为 $O(n^2)$，与边数无关，对稠密图比较有利。

算法结束后，minedge 中权为负的边就是生成树的 n−1 条边。通过边集表示生成树，可直接表达结点间的连通关系，当然也表达了树的全部信息。

上述算法中查找最短边的部分还可进行修改，如将已生成的边移到数组的前面，以后可只在数组的后面找最短边，可提高查找效率（但这时边的序号不能反映边的起点，需要在边的存储结构中增加起点信息）；又如采用小根堆（见第 7 章堆排序）来找最短边，查找的效率还可提高到 $O(n\log_2 n)$。但调整部分不变，总的时间复杂度仍为 $O(n^2)$。

6.5.2　Kruskal 算法

Prim 算法每一次都从连接 U 与 V−U 的候选边中选最小边，但它不一定是所有当前未选用的但属于最终最小生成树的边中最小者，因为此时还有两个端点都在 V−U 的边没有考虑。换言之，Prim 算法不是按边权递增的次序生成最小生成树的。

构造最小生成树的另一个算法是**克鲁斯卡尔**（Kruskal）提出的，它的基本思想是按边权递增次序生成最小生成树。即若某边是最小生成树中第 i 小的边，则它在第 1～第(i−1)小的边全部选出后才加入到中间的部分结果中。

Kruskal 算法的基本过程为：设 G=(V, E)是连通网络，令最小生成树的初始状态为只有 n 个顶点而无边的非连通图 T=(V, ∅)，T 中每个顶点自成一个连通分量。以后按长度递增的顺序依次选择 E 中的最短边(u, v)，若其端点 u、v 分别属于当前 T 的两个连通分量 T_1、T_2，则将该边加入到 T 中，T_1 和 T_2 也由此边连成一个连通分量；若 u、v 属于当前同一个连通分量，则舍去此边（因为每个连通分量都是一棵树，此边添加到树中将形成回路）。依次类推，直到 T 中所有顶点都属于同一个连通分量为止，T 便是 G 的一棵最小生成树。

对图 6.16（a）所示的连通网络，按 Kruskal 算法构造最小生成树，其过程如图 6.18 所示。第一次取最短边(1, 3)，它连接两个不同的连通分量，故将它加到 T；类似，依次取边(4, 6)、(3, 6)、(2, 5)，将它们加到 T 中，如图 6.18（b）～（d）所示。接着考虑当前最短边(1, 4)、(2, 3)和(3, 5)，它们的长度相同，但边(1, 4)的两个端点属于同一个连通分量，舍去。边(2, 3)和(3, 5)都符合要求，这里不妨选择边(2, 3)加入 T，见子图（e）。子图（e）为单个连通分量，它就是所求的一棵最小生成树。如果在子图（d）中选择边(3, 5)而不是边(2, 3)，将它添加到当前的 T 中，则得到另一棵如图 6.17 所示的最小生成树。

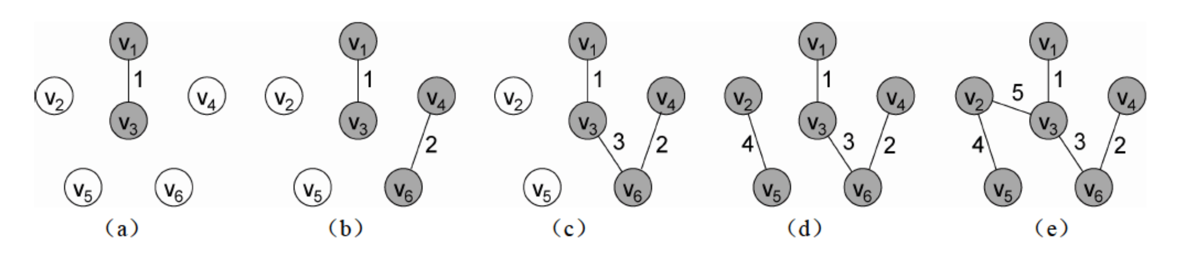


图 6.18　Kruskal 算法构造最小生成树的过程

由于每个连通分量都是一棵树，它们构成森林（初始有 n 棵树），随着算法的进行，森林中的树逐步连通（合并），最后合并为一棵树，即为所求得的最小生成树。可见，Kruskal 算法是按边权递增的次序连通森林的。下面给出 Kruskal 算法的粗略描述：

```
T=(V,∅);
while(T 中所含边数<n-1) {
  从 E 中选取当前最短边(u,v),并从 E 中删去之;
  if(边的端点 u,v 属于不同连通分量) {将边(u,v)并入 T 中;合并两个连通分量;}
}
```

Kruskal 算法中要判断边的两个端点是否属于同一连通分量，以及合并两个不同的连通分量[①]。对同一连通分量的判断若是从边的一个端点出发遍历搜索另一个端点，则显然效率很低。一个简单方法是设置一个辅助数组 sets[1..n]，记录每个顶点的连通分量号，初始时 sets[i]=i。这样，对任一边(v_i, v_j)，若 sets[i]=sets[j]，则两端点在同一连通分量。但每次合并两个连通分量时，要扫描辅助数组，将其中一个连通分量中所有点的连通分量号改为另一个，运算量为 O(n)。

较好的方法是将各个连通分量用树来记录：开始时，每个树只有一个根；以后每次合并两个连通分量时，就将两棵树合并为一棵树，这只要简单地使其中一个树根成为另一个树根的孩子即可。这样，检查两个端点是否属于同一个连通分量，只要检查它们所在树的根是否相同。为此，每个结点需要设置双亲指针，以便能沿双亲指针找到根。显然，找到根的效率取决于树的高度。

为了降低树的高度，可在找根的过程中“顺便”把路径上各结点的双亲指针往上提，如改为指向祖父结点或者干脆指向根。这个过程可减少以后查找的结点数，但增加了修改指针的开销。另一个常用的方法是，在合并树时，令结点数少的树作另一个树的子树。这需要在树根中记录结点数，但不必采用专门的存储空间，可将结点数以负数的形式记录在原树根的双亲域中，约定只要双亲域为负，就表示它为树根。这时树的高度最大为 $\lfloor \log_2 n \rfloor+1$[②]，每次合并的运算量是 $O(\log_2 n)$。最后算法如下：

```
typedef struct {
  int v1,v2;
  int len;
} edgetype;                    //边的类型：两个端点号和边长
int parent[nmax+1];            //结点的双亲指针数组，设为全局量，nmax 为结点数最大值
int getroot(int v) {           //找结点 v 所在的树根
  int i;
  i=v;
  while(parent[i]>0) i=parent[i];
  return i;                    //若无双亲(初始点)，双亲运算结果为其自己
}
int getedge(edgetype E[],int e) {……}//找最短边（内容略）
void kruskal(edgetype E[],int n,int e) {//n 为结点数，e 为边数
  int i,p1,p2,m,i0;
```

① 类似问题的一般模型是**并查集**：集合的主要运算是“并”（集合的合并）、“查”（查找元素所属的集合）。具体实现时可把集合的元素组织成数组、链表或树等，其中组织成树时效果较好。

② n=1 时显然成立。设 i≤n−1 时成立，则 i=n 时，设最后合并的两个树为 j、k，结点数分别为 m 和 n−m，不妨设 1≤m≤n/2，则 j 成为 k 的子树。于是新树的高度要么与 k 的相同，要么比 j 的大 1。对前者，新树高度≤$\lfloor \log_2(n-m) \rfloor+1$≤$\lfloor \log_2 n \rfloor+1$；对后者，新树高度≤$\lfloor \log_2 m \rfloor+2$≤$\lfloor \log_2 n/2 \rfloor+2$≤$\lfloor \log_2 n \rfloor+1$。

```
  for(i=1;i<=n;i++)
    parent[i]=-1;          //每个初始连通分量只有一个根结点，无双亲(双亲置负)
  m=1;
  while(m<n) {
    i0=getedge(E,e);       //获得最短边号
    p1=getroot(E[i0].v1);
    p2=getroot(E[i0].v2);
    if(p1==p2) continue;   //连通分量相同，不合并
    if(-p1<-p2) {          //p1 的结点数较少
      parent[p2]=parent[p1]+parent[p2]; //p2 的双亲中累计结点总数(为负值)
      parent[p1]=p2;       //p1 作为 p2 的子树
    }
    else {
      parent[p1]=parent[p1]+parent[p2];
      parent[p2]=p1;
    }
    m++;
    cout<<"The Edge "<<m<<" is: "<<E[i0].v1<<" "<<E[i0].v2<<endl;
  }
}
```

算法初始化的时间为 $O(n)$；找最短边时，若是简单地对各边进行扫描，每次时间为 $O(e)$，找全部最短边的时间则为 $O(e^2)$，但若把各边组织成小根堆（见第 7 章堆排序），则找最短边的时间除了第一次为 $O(e)$外，其他每次都不超过堆的深度 $O(\log_2 e)$，故找全部最短边的时间不超过 $O(e\log_2 e)$；判断连通分量的时间不超过树的高度 $O(\log_2 n)$，对连通分量的全部判断时间不超过 $O(e\log_2 n)$；连通分量合并 n−1 次，时间为 $O(n)$。所以总时间不超过 $O(n)+O(e\log_2 e)+O(e\log_2 n)+O(n)$。由于连通图的边数 $e \geqslant n-1$，所以 $O(n) \leqslant O(e)$，$O(\log_2 n) \leqslant O(\log_2 e)$，最后总的时间复杂度为 $O(e\log_2 e)$。

可见，Kruskal 算法的时间复杂度与网络中边的数目 e 有关，对求稀疏图的最小生成树比较有利。

6.6 最短路径

路径问题是图中的又一基本问题，很多实际问题都可抽象或归纳为最短路径问题。比如交通网络中常常提出这样的问题：两地之间是否有路相通？在有多条通路的情况下，哪一条最短？交通网络可以用带权图表示，图中顶点表示城镇，边表示两个城镇之间的道路，边上权值可表示两城镇间的距离、交通费用或途中所需的时间等。以上提出的问题就是带权图中求最短路径的问题，即求两个顶点间长度最短的路径。这里路径长度不是指路径上边的个数，而是指路径上各边的权值总和，它的具体含义取决于边上权值所代表的意义。

两地之间由于上坡、下坡，顺水、逆水等不同，来回所花的时间一般也会不同。考虑到交通网络的这种有向性，本节只讨论有向网络的最短路径问题。习惯上称路径的开始顶点为**源点**（Source），路径的最后一个顶点为**终点**（Destination）。为了讨论方便起见，设顶点集 V={1, 2, …, n}，并假定所有边上的权值均是表示长度的非负整数。

6.6.1 单源最短路径

单源（Single−source）最短路径问题是：对于给定的有向网络 G=(V, E)及单个源点 v，求 v 到 G 的其余各顶点的最短路径。也许实际问题只关心某两点间的最短路径，但遗憾的是，求某两点间的最短路径并不比求其中一点到其他所有点的最短路径有更好的算法。关于这类问题，荷兰籍计算机科学家 Dijkstra 在 1959 年首先给出了一个成熟的算法，一般称之为 Dijkstra 单源最短路径算法。

在有向网络中，从某点出发，到达其他任何一点都可能有多条路径，其中必有一条最短路径（若没有路径，则假设路径是长度为无穷大的虚拟路径）。设图共有 n 个顶点，于是从某一点出发到其他各点的最短路径有 n−1 条。这 n−1 条最短路径之间也存在大小关系。例如，对图 6.19 所示的有向网络，n=5，若求顶点 1 到其他各顶点的最短路径，则有 n−1=4 条，将它们按长度递增的次序排列，见表 6.2。

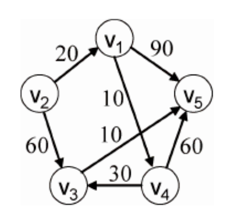


图 6.19 有向网络

表 6.2 图 6.19 中源点 1 到各顶点的最短路径

源点	中间顶点	终点	路径长度
1		4	10
1	4	3	40
1	4, 3	5	50
1		2	∞

从图 6.19 及表 6.2 可见，如果按长度递增的次序生成源点 v 到其他顶点的最短路径，则当前正在生成的顶点 w 的最短路径上除终点以外，其余中间顶点的最短路径均已生成(因为它们的最短路径长度比当前顶点 w 的最短路径短)。

Dijkstra 算法就是按长度递增的次序生成各顶点的最短路径，即先求出长度最小的一条最短路径，然后求出长度第二小的最短路径，……，最后求出长度最大的最短路径。算法的基本思想是，设置并逐步扩充一个集合 S，存放已求出其最短路径的顶点，则尚未确定最短路径的顶点集合是 V−S。为称呼方便，以下把源点 v 到终点 w 的最短路径简称为顶点 w 的最短路径，集合 S 中的顶点称为已求点，集合 V−S 中的顶点称为待求点。算法开始时，S 中已求点就是源点自己，以后每一步就是按最短路径长度递增的顺序在待求点集合中选一个路径长度最短的点来扩充已求点集 S，直到所有顶点都成为已求点。

但是，待求点的最短路径长度本身就是待求的，又如何找出其中的最短呢？观察表 6.2 可得启发：这种待求点的最短路径上，除终点外，其余顶点都是已求点。于是，对图中每一顶点 i，必须记住从源点 v 到 i、且中间只经过已求点的最短路径长度。为称呼方便，将此长度称为顶点 i 的**距离值**。为此，定义一个数组 D[1..n]来存放各顶点的距离值。

显然，已求点的距离值就是该点的最短路径长度，而待求点的距离值则不一定是该点的最短路径长度，因为从源点到该待求点可能存在经过其他待求点的更短路径。但可以证明，若当前待求点中距离值最小的点为 k，则其距离值 D[k]就是 k 点的最短路径长度，并且 k 也是待求点中最短路径长度最短的顶点。如图 6.20 所示，这两点可简单地证明如下：

（1）任取一条从源点 v 到 k 点的路径 P_{vk}，该路径可能经过若干待求点，设经过的第一

个待求点为 x，则该路径可分为两段 P_{vx} 和 P_{xk}，于是：

P_{vk} 长度≥P_{vx} 长度

≥x 距离值（因为 P_{vx} 中间只经过已求点，而距离值是所有这类路径中最短的）

≥k 距离值（因为 k 点是所有待求点中距离值最小的）

即 k 点的距离值就是 k 点的最短路径长度。

（2）对任意一个待求点 j，任取一条从源点 v 到顶点 j 的路径 P_{vj}，该路径也可能经过若干待求点，设经过的第一个待求点为 y，则该路径可分为两段 P_{vy} 和 P_{yj}，于是与（1）类似，有

P_{vj} 长度≥P_{vy} 长度

≥y 距离值

≥k 距离值

即 k 点的距离值也是所有待求点中路径长度最短的。

由上面两点，我们便找到了扩充已求点集 S 的方法，即每一步在当前待求点集中选择一个距离值最小的点 k 扩充到 S 中。此时，已求点集 S 增加了一个点 k，剩下的待求点的距离值可能发生变化（即减少），因为这时增加了经过新已求点 k 的各种可能路径，所以必须调整其余各待求点的距离值。

下面的问题是如何调整剩余待求点的距离值。注意到 k 点加入到已求点集 S 后，若某个待求点 j 的距离值有变化（即减少），则对应的路径必定是从源点 v 途经 k 最后到达待求点 j 的路径 P_{vkj}。由于 P_{vkj} 中间只经过已求点，它的前一段 P_{vk} 必定是 k 的最短路径，其长度为 D[k]；它的后一段 P_{kj} 只有两种可能：其一是由 k 经过边<k, j>直达待求点 j，其二是从 k 出发再经过若干已求点后到达 j。但后一种情形是不可能的，如图 6.21 所示，有：



图 6.20　路径 P_{vxk}　　　　**图 6.21　路径 P_{vkxj}**

P_{vkxj} 长度=P_{vk} 长度+P_{kx} 长度+P_{xj} 长度

≥k 距离值+P_{kx} 长度+P_{xj} 长度

≥x 距离值+P_{kx} 长度+P_{xj} 长度（因为 x 比 k 先加入已求点集 S，其距离值较小）

≥x 距离值+P_{xj} 长度（因为 P_{kx} 长度≥0）

=P_{vxj} 长度

≥j 距离值（因为 P_{vxj} 中间只经过已求点，而距离值是所有这类路径中最短的）

即这种情形下距离值不可能减少。所以需要按第一种情形调整距离值，即对待求点集扫描检查，若某点 j 的原距离值大于新路径长度，即 D[k]+边<k, j>的权，则将 D[j]修改为此长度。

上述算法是一种贪心算法，它每次都从待求点中得到一个离源点最近的点。对图 6.19

所示的有向网络，按该算法求源点 1 到其余各顶点的最短路径的过程，见图 6.22。图中阴影圈表示已求点，其他为待求点，圆圈旁的数字表示该点当前的距离值，连接两个已求点的边用实线表示，连接已求点和待求点的边用虚线表示。

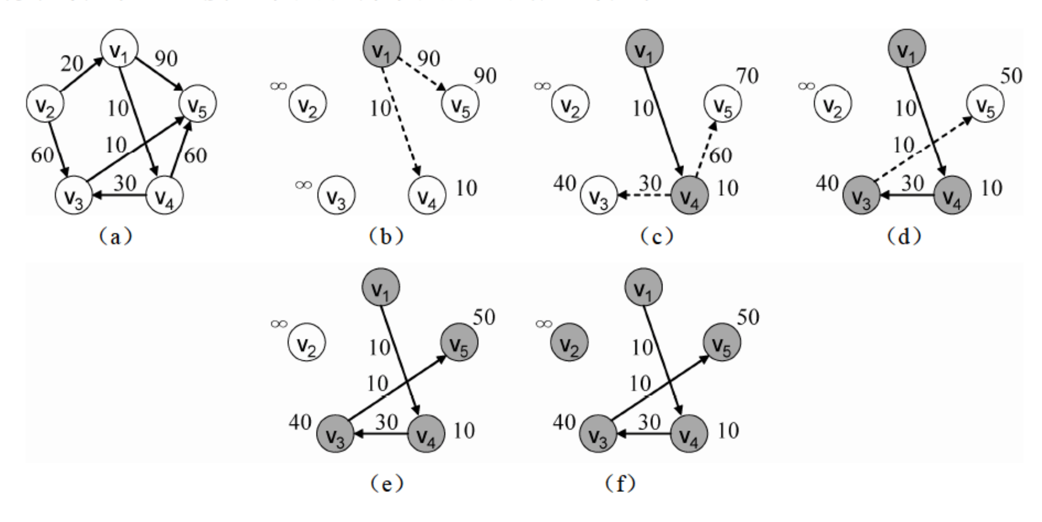


图 6.22　Dijkstra 算法求源点 1 到其余各顶点的最短路径

开始时，已求点集 S 只有一个源点 1，初始待求点 j（j=2, …, 5）的距离值 D[j]即有向边<1, j>的权，见子图（b）。其中边<1, 2>和<1, 3>不存在，故 D[2]=∞，D[3]=∞，可见待求点 4 的距离值 D[4]=10 最小，它就是源点 1 到顶点 4 的最短路径，将顶点 4 加入已求点集 S。此时，待求点 3 有一条新路径 P_{143}，长度为 40，小于原距离值∞；待求点 5 也有一条新路径 P_{145}，长度为 70，小于原距离值 90，故分别将待求点 3 和 5 的距离值调整为 40 和 70；待求点 2 的距离值不变，见子图（c）。类似进行下去，可依次求得顶点 3, 5, 2 的最短路径及其长度，如子图（d）、（e）和（f）所示。

一般求最短路径长度时，还要知道具体路径，为此可设置一个路径向量 P[1..n]，其中 P[i]表示 i 点的最短路径上该点的前趋顶点。这样可根据 P 找到路径上每个顶点的前趋，从而得到最短路径。设有向网络以邻接矩阵存储，具体算法如下：

```
void dijkstra(mat_graph *G,int v,int D[],int P[],char S[]) {
//v 为源点,D 为距离值数组，P 为路径前趋数组，S 为已求点标志数组。所有数组都从下标 1 开始使用
  int i,j,k,pre,min;
  for(i=1;i<=G->n;i++) {           //初始化距离值、前趋、已求点集
    S[i]=0;
    D[i]=G->adjmat[v][i];
    P[i]=v;
  }
  S[v]=1;D[v]=0;                   //将源点 v 放入 S
  for(i=1;i<G->n;i++) {
    min=INT_MAX;                   //INT_MAX 为整数最大值，表示∞
    k=0;                           //k 记录最短距离值点(此处可不必设初值)
    for(j=1;j<=G->n;j++)           //找距离值最小的待求点
      if(!S[j] && D[j]<min) {
        min=D[j];
        k=j;
      }
    if(min==INT_MAX) break;        //剩余点距离值都为∞，不必调整距离值
```

```
      S[k]=1;                          //将 k 加入 S
      for(j=1;j<=G->n;j++)             //调整剩余点的距离值和前趋
        if(!S[j] && D[j]>D[k]+G->adjmat[k][j]) {
          D[j]=D[k]+G->adjmat[k][j];
          P[j]=k;
        }
    }
    for(i=1;i<=G->n;i++) {
      cout<<D[i]<<": "<<i;
      pre=i;
      do {
        pre=P[pre];
        cout<<"<-"<<pre;
      } while(pre!=v);
      if(D[i]==INT_MAX) cout<<"(无路径)"
      cout<<endl;
    }
  }
```

注意数组从下标 1 开始使用，所以有关的 C/C++语言数组在定义时应多一个元素，如 D[n+1]、P[n+1]、S[n+1]等，其中 S 为已求点集数组，实际只需设置一个已求标志即可，故取为 char 类型以节省空间。特别地，注意到图 G 邻接矩阵的对角元素一般没有什么作用，还可用对角元素 G–>adjmat[i][i]来充当 S[i]。

对图 6.19 所示的有向网络，若源点为 1，则算法执行过程中 D、P 等的变化情况如表 6.3 所示，其中阴影部分为已求出的最短路径长度。

表 6.3　算法 dijkstra 的动态执行情况

循环	已求点	k	D[1]	D[2]	D[3]	D[4]	D[5]	P[1]	P[2]	P[3]	P[4]	P[5]
初始化	{1}	—	0	∞	∞	10	90	1	1	1	1	1
1	{1, 4}	4	0	∞	40	10	70	1	1	4	1	4
2	{1, 4, 3}	3	0	∞	40	10	50	1	1	4	1	3
3	{1, 4, 3, 5}	5	0	∞	40	10	50	1	1	4	1	3
4	{1, 4, 3, 5, 2}	2	0	∞	40	10	50	1	1	4	1	3

注意，最后一行实际并未执行（min=∞，提前结束）。算法最后打印输出的结果为：

```
0:  1<-1
∞: 2<-1(无路径)
40: 3<-4<-1
10: 4<-1
50: 5<-3<-4<-1
```

显然，若输出某顶点的最短路径长度为∞，则从源点到该顶点没有路径；而源点到自己的最短路径长度是 0。

容易看出，算法 dijkstra 的时间复杂度为 $O(n^2)$，占用的辅助空间是 O(n)。由于时间复杂度与边数基本无关，所以比较适合稠密图。

上述算法求最小距离值的时间还可以减少，即不对所有待求点搜索，而采用后面第 7 章将介绍的小根堆，并将各待求点较小的新距离值添加到堆中再重建堆（不删除原距离值，否则处理困难）。但剩余点距离值调整的时间不变，所以算法的时间复杂度仍为 $O(n^2)$。

6.6.2 所有顶点对之间的最短路径

所有顶点对（All−pairs）之间的最短路径问题是：对于给定的有向网络 G=(V, E)，要求对 G 中任意两个不同的顶点 v、w（v≠w），找出 v 到 w 的最短路径。

显然，如果依次把有向网络的每个顶点作为源点，重复执行 dijkstra 算法 n 次，就可求得每对顶点之间的最短路径，其时间复杂度为 $O(n^3)$。对此问题，Floyd 在 1962 年发现了一个更为直接的算法，不过它的时间复杂度也是 $O(n^3)$。

Floyd 算法实质上是一种迭代法，它在图的邻接矩阵上做 n 次迭代。第 n 次迭代后，邻接矩阵上第 i 行第 j 列的元素值即为 i 到 j 的最短路径值。

为了便于理解 Floyd 算法，先从直观上进行分析。对 1⩽i, j⩽n，若 i 到 j 有边，则它是 i 到 j 的一条路径，长度为 A[i][j]。但它不一定是 i 到 j 的最短路径，因为可能存在一条从 i 到 j，中间经过其他顶点的路径。中间点可能为 1, 2, …, n 中的一个或多个，如果不采用某种方法系统地进行处理，势必太烦琐。Floyd 把这个过程递推地进行，即先考虑中间点序号不大于 k 的情况，然后考虑中间点序号不大于 k+1 的情况，具体地说就是：

（1）首先，考虑从顶点 i 到 j 是否有中间点序号不大于 1 的最短路径。这一步实际上是考虑是否有以顶点 1 为中间点的路径<i, 1, j>，即考虑 G 中是否有边<i, 1>和<1, j>，若有，则该路径的长度为 A[i][1]+A[1][j]，这时，比较路径<i, j>和<i, 1, j>的长度，取较短者为当前最短路径。

（2）其次，考虑从顶点 i 到 j 是否有中间点序号不大于 2 的最短路径。即是否有路径<i, …, 2, …, j>，若没有，则当前最短路径不变；若有，则该路径长度为路径<i, …, 2>的长度+路径<2, …, j>的长度，而这两条路径就是前一步求出的中间点序号不大于 1 的最短路径。将新的路径长度和原来的路径长度作比较，取较短者为当前最短路径。

（3）然后，再考虑从顶点 i 到 j 是否有中间点序号不大于 3 的最短路径。依次类推，直到考虑到中间点序号不大于 n 的最短路径后，便考虑完了中间点为任何点的可能性，故最终得到 i 到 j 的最短路径。

实现上述算法的关键，是保留每一步求得的所有顶点对之间的当前最短路径长度。为此，我们需要一个 n×n 的方阵序列 A_i[1..n][1..n]（i=0, 1, …, n）来保存每一步的结果，其中 A_k[i][j]表示从 i 到 j 中间点序号不大于 k（0⩽k⩽n）的最短路径长度。

显然，A_0 就是 G 的邻接矩阵，它表示任一顶点对之间不经过任何中间点的最短路径长度。Floyd 算法的基本思想是，从 A_0 开始，递推地产生矩阵序列 $A_1, A_2, \cdots, A_n$。

现在假设 A_{k-1} 已求得，如何递推求出 A_k 呢？注意到在第 k 步，对于任意一对顶点 i, j，从 i 到 j 中间点序号不大于 k 的最短路径只有两种情况：

（1）中间不经过顶点 k，那么有：

$A_k[i][j]=A_{k-1}[i][j]$

（2）中间经过顶点 k，则该路径由两段路径<i, …, k>和<k, …, j>组成，它们都是前一步求出的中间结点序号不大于 k 的最短路径，于是：

$A_k[i][j]=A_{k-1}[i][k]+A_{k-1}[k][j]$

综合上面的结果，可得 A_k 的递推公式：

$A_0[i][j]=A[i][j]$

$A_k[i][j]=\min(A_{k-1}[i][j], A_{k-1}[i][k]+A_{k-1}[k][j])$ $1\leqslant i, j\leqslant n，1\leqslant k\leqslant n$

上述算法需要一个矩阵序列 $A_0, A_1,\cdots, A_n$，但实际上只用一个矩阵就可，即在同一个矩阵中进行迭代。这是因为，在第 k 次迭代时，$A_{k-1}[i][k]$和 $A_{k-1}[k][j]$的值不变。以 $A_k[i][k]$为例，它表示从 i 到 k 中间点序号不大于 k 的最短路径。该路径上如果不含顶点 k，则路径上中间顶点的序号不大于 k−1，于是 $A_k[i][k]=A_{k-1}[i][k]$；如果含有顶点 k，则该路径为 <i, ⋯, k, ⋯, k>，前一段路径<i, ⋯, k>的长度仍是 $A_{k-1}[i][k]$，后一段路径<k, ⋯, k>的长度非负，则该路径不可能变短。所以总有 $A_k[i][k]=A_{k-1}[i][k]$。由于 A[i][k]和 A[k][j]的值在当前迭代中不变，于是计算 $A_k[i][j]=A_{k-1}[i][k]+A_{k-1}[k][j]$时便可在同一个矩阵中进行。

为了得到最短路径本身，设置一个路径矩阵 path[1..n][1..n]，它也是迭代产生的，其作用类似于 Dijkstra 算法中的路径数组，但这里 $path_k[i][j]$存放 i 到 j 的中间点序号不大于 k 的最短路径上顶点 i 的后继顶点（当然也可存放顶点 i 的前趋顶点）。算法结束时，由 path[i][j]即可找到 i 到 j 的最短路径上的各个顶点。

具体算法如下：

```
void floyd(mat_graph *G) {//A 为最短路径值矩阵，path 为最短路径矩阵，假设为全局变量
  int i,j,k,next;
  for(i=1;i<=G->n;i++)            //初始化 A 和 path
    for(j=1;j<=G->n;j++) {
      A[i][j]=G->adjmat[i][j];
      path[i][j]=j;               //j 是顶点 i 的后继
    }
  for(k=1;k<=G->n;k++)            //n 次迭代
    for(i=1;i<=G->n;i++)
      for(j=1;j<=G->n;j++)
        if(A[i][k]+A[k][j]<A[i][j]) {
          A[i][j]=A[i][k]+A[k][j];        //修改路径长度
          path[i][j]=path[i][k];          //修改路径后继
        }
  for(i=1;i<=G->n;i++)            //输出所有顶点对之间的最短路径及长度
    for(j=1;j<=G->n;j++) {
      cout<<A[i][j]<<": "<<i;     //输出路径长度
      next=i;
      do {
        next=path[next][j];       //找后继点
        cout<<"->"<<next;         //输出后继顶点
      } while(next!=j);
      if(A[i][j]==INT_MAX) cout<<"(无路径)";//INT_MAX 为整数最大值，表示∞
      cout<<endl;
    }
}
```

显然该算法时间复杂度为 $O(n^3)$，比较适合稠密图。注意，算法中有关数组下标从 1 开始使用（以适应从 1 开始的顶点编号），所以在定义有关的 C/C++二维数组时应多一行一列，如迭代矩阵 A[n+1][n+1]和路径矩阵 path[n+1][n+1]。

以图 6.19 所示的有向网络为例，上述 Floyd 算法在迭代过程中矩阵 A 和 path 的变化

以及最终结果如下：

$$A_0=\begin{bmatrix}0&\infty&\infty&10&90\\20&0&60&\infty&\infty\\\infty&\infty&0&\infty&10\\\infty&\infty&30&0&60\\\infty&\infty&\infty&\infty&0\end{bmatrix}\qquad path_0=\begin{bmatrix}1&2&3&4&5\\1&2&3&4&5\\1&2&3&4&5\\1&2&3&4&5\\1&2&3&4&5\end{bmatrix}$$

$$A_1=\begin{bmatrix}0&\infty&\infty&10&90\\20&0&60&30&110\\\infty&\infty&0&\infty&10\\\infty&\infty&30&0&60\\\infty&\infty&\infty&\infty&0\end{bmatrix}\qquad path_1=\begin{bmatrix}1&2&3&4&5\\1&2&3&1&1\\1&2&3&4&5\\1&2&3&4&5\\1&2&3&4&5\end{bmatrix}$$

$A_2=A_1$　　　　path$_2$=path$_1$

$$A_3=\begin{bmatrix}0&\infty&\infty&10&90\\20&0&60&30&70\\\infty&\infty&0&\infty&10\\\infty&\infty&30&0&40\\\infty&\infty&\infty&\infty&0\end{bmatrix}\qquad path_3=\begin{bmatrix}1&2&3&4&5\\1&2&3&1&3\\1&2&3&4&5\\1&2&3&4&3\\1&2&3&4&5\end{bmatrix}$$

$$A_4=\begin{bmatrix}0&\infty&40&10&50\\20&0&60&30&70\\\infty&\infty&0&\infty&10\\\infty&\infty&30&0&40\\\infty&\infty&\infty&\infty&0\end{bmatrix}\qquad path_4=\begin{bmatrix}1&2&4&4&4\\1&2&3&1&3\\1&2&3&4&5\\1&2&3&4&3\\1&2&3&4&5\end{bmatrix}$$

$A_5=A_4$　　　　path$_5$=path$_4$

```
0: 1->1
∞: 1->2(无路径)
40: 1->4->3
10: 1->4
50: 1->4->3->5
20: 2->1
0: 2->2
60: 2->3
30: 2->1->4
70: 2->3->5
…
0: 5->5
```

最后指出，在以上最短路径问题中，要求边的权值非负。若边的权值可正可负，则单源路径的 Dijkstra 算法不再有效（可能得到错误结果），需采用其他算法，如 Bellman−Ford 算法（要求权值为负的边不出现在某个回路），而所有点对路径的 Floyd 算法仍可使用（也要求权值为负的边不出现在某个回路）。显然，对无向图，若可把每条边拆成反向的两条有向边，则前述算法也可适用。



6.7　有向无环图及其应用

无回路的有向图称为**有向无环图**（Directed Acyclic Graph，DAG）。我们在广义表中，就曾用 DAG 来描述有共享成分的再入表。在工程应用上，有向无环图是描述一项工程或系统进行过程的有效工具。本节介绍这方面的两个应用，它们对应这样的两个问题：

（1）整个工程能否顺利进行？

（2）完成整个工程的最短时间是多少？哪些活动是影响整个工程进度的关键？

6.7.1　拓扑排序

在现实生活中，一个大的**工程**（Project）常常可分成若干较小的子工程，当所有子工程都完成时，整个工程也就完成了。子工程简称**活动**（Activity），它们之间一般有两种关系：

（1）先后关系（依赖关系），一个活动完成后，另一个活动才能开始。

（2）并列关系（独立关系），活动可独立、并行地进行，互不影响。

如何安排各个活动的先后顺序，使各个活动都能顺利进行呢？这类有关工程进度、次序规划之类的问题，都可归结到拓扑排序。拓扑排序是定义在有向图上的一种操作，目的是根据顶点间的关系求得顶点的一个线性排列。

例如，假设计算机专业的学生要学完表 6.4 所列出的课程。在这种情况下，工程就是完成给定的学习计划，而活动就是学习一门课程。其中，有些课程是基础课，不需要先修其他课程，如《高等数学》；另一些课程则必须在学完某些先修课程后才能开始学习，如学习《数据结构》之前，必须先学完《高级语言程序设计》。

表 6.4　课程进程关系

课 程 编 号	课 程 名 称	先　修　课
c_1	高等数学	–
c_2	高级语言程序设计	–
c_3	数据结构	c_2
c_4	编译原理	c_2，c_3
c_5	操作系统	c_3，c_6
c_6	计算机组成原理	c_7
c_7	普通物理	c_1

所有活动之间的关系可用有向图表示：顶点表示活动，有向边表示活动之间的先后关系，即如果有边<i, j>，则表示活动 i 完成后活动 j 才能开始。图 6.23 的有向图就表示了表 6.4 中各课程间的先后关系。

一般地，我们把顶点表示活动、边表示活动之间先后关系的有向图，称为**顶点表示活动的网**（Activity On Vertex network，AOV 网）。

对于一个有向图，常常要将它的所有顶点排成一个线性序列 v_1，v_2，…，v_n，满足下述

关系：若图中从顶点 v_i 到顶点 v_j 有路径，则在该序列中 v_i 必须排在 v_j 之前，否则 v_i 与 v_j 的次序任意。这种序列称为**拓扑序列**，构造拓扑序列的操作称为**拓扑排序**（Topological Sorting）。

例如，对图 6.23 所示的有向图进行拓扑排序，可以得到拓扑序列：$c_1, c_2, c_7, c_6, c_3, c_4, c_5$ 和 $c_1, c_7, c_2, c_3, c_6, c_4, c_5$ 等（还有多个）。如果一个学生每个学期只能修读一门课程，那么该生按照某个拓扑序列的次序学习就可保证任一课程的正常学习，即先修课程已学完。如果每学期需修多门课，则可对拓扑序列作些修改，识别出哪些课可同时修读即可。

一般情况下，假设有向图代表一个工程计划，若条件限制只能串行工作，则该图的一个拓扑序列就是整个工程得以顺利完成的一种可行方案。并非任何有向图的顶点都可以排成拓扑序列，若图中存在有向回路时就没有。例如，图 6.24 的有向图中存在一个有向回路 v_3，v_6，v_5，其中任一顶点都有路径到达另外两个顶点，于是任一顶点都要排在另外两个顶点之前，这是不可能的。通常，表示某项实际工程计划的有向图是不应该存在有向回路的，因为出现回路意味着：某些活动的开工将以自己工作的完成为先决条件，这种现象称为**死锁**，此项工程是不可行的。

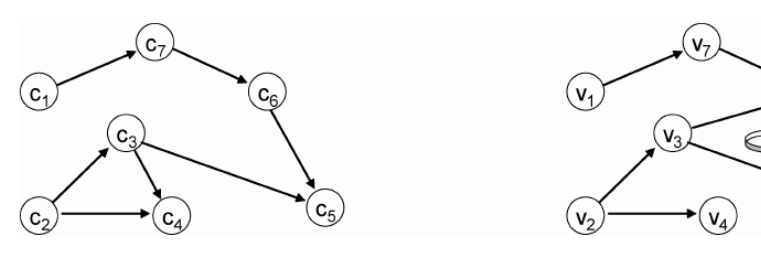


图 6.23　课程间的先后关系　　　　图 6.24　有向图的死锁现象

任何 DAG 的顶点都可以排成一个拓扑序列，但拓扑序列不一定唯一。例如，前面的例子就给出了两个拓扑序列，而且还可以构造出更多的拓扑序列。

拓扑排序的基本方法是简单而直观的，其基本步骤如下：

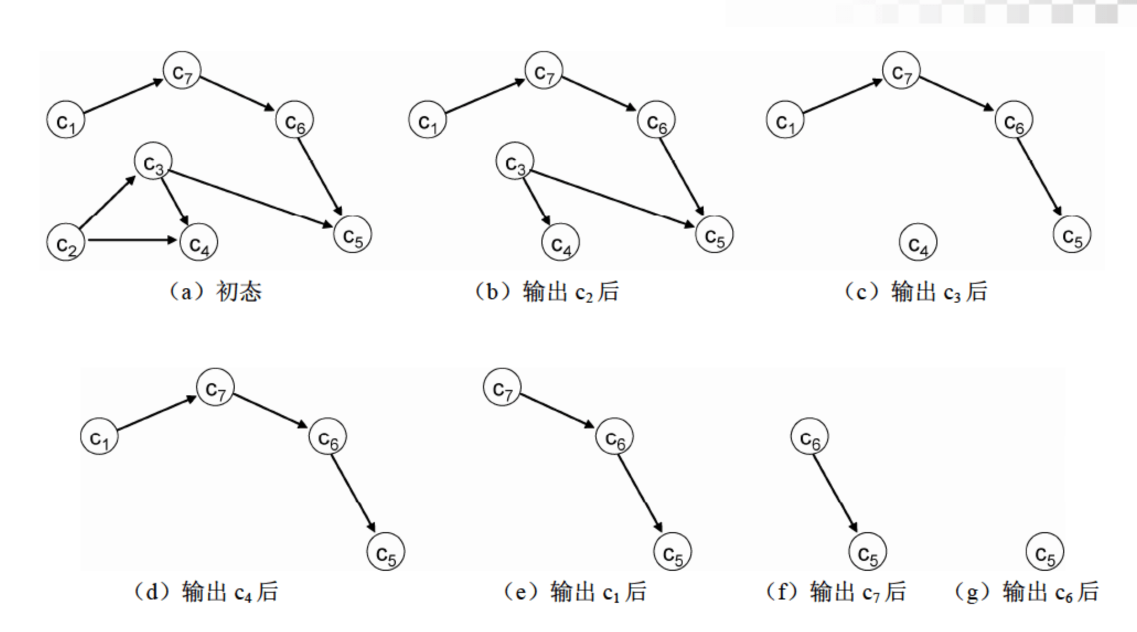
（1）从图中选择一个入度为 0 的顶点且输出之。

（2）从图中删掉此顶点及其所有出边。

反复执行这两步，直至所有顶点都已输出，此时整个拓扑排序完成；或者直到图中剩余顶点的入度都不为 0 时终止，此时图中存在回路，拓扑排序不能再进行下去。

以图 6.23 为例，拓扑排序过程如图 6.25（b）～图 6.25（g）所示。刚开始时，c_1 和 c_2 的入度为 0，假设先输出序号较大的，即 c_2，在删除 c_2 及其出边 $<c_2, c_3>$ 和 $<c_2, c_4>$ 后得到子图（b）。此时入度为 0 的顶点是 c_1 和 c_3。假设输出 c_3，删除其出边 $<c_3, c_4>$ 和 $<c_3, c_5>$，得到子图（c）。依此类推，直至得到子图（g）之后，图中仅有一个入度为 0 的顶点 c_5，输出后整个拓扑排序结束。最后得到的拓扑序列是：$c_2, c_3, c_4, c_1, c_7, c_6, c_5$。

下面以邻接表作存储结构，讨论拓扑排序算法的实现。为了便于考察每个顶点的入度，在顶点表中增加一个入度域 in，以指示当前各个顶点的入度值。每个顶点的初始入度值可在邻接表的生成过程中累计得到。例如图 6.25（a）的有向图，其邻接表如图 6.26 所示。增加入度域后的顶点表如下：



（a）初态　（b）输出 c_2 后　（c）输出 c_3 后

（d）输出 c_4 后　（e）输出 c_1 后　（f）输出 c_7 后　（g）输出 c_6 后

图 6.25　拓扑排序过程

```
typedef struct {
  datatype data;      //顶点信息
  int in;             //入度域
  pointer first;      //边表头指针
} headtype;           //顶点表结点类型
```

在算法的第一步，找入度为 0 的顶点只要对顶点表的入度域扫描即可。为了避免每次都在所有顶点中重复寻找入度为 0 的点，可以设置一个栈来保存当前所有入度为 0 的点，以后每次选入度为零的顶点时，直接从栈顶取出。排序过程中一旦出现新的入度为 0 的点，就将其入栈。在拓扑排序之前，对顶点表扫描一遍，将所有初始入度为零的点入栈。

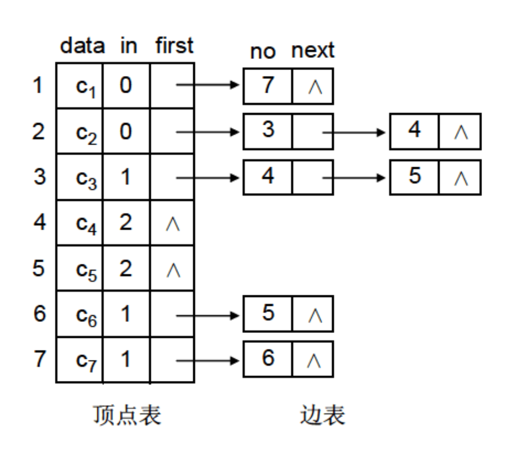


图 6.26　图 6.25(a)带入度域的邻接表

算法的第二步，是删去已输出的顶点及以该顶点为起点的出边，其目的是要改变这些出边上终点的入度。因此，只要检查从栈顶弹出的点（相当于删去此点）的出边表，把每条出边的终点所对应的入度值减 1（相当于删去出边），就完成了第二步操作。

根据上面的叙述，我们得到以邻接表作存储结构的拓扑排序算法，其概要描述如下：

```
void topsort(graph g) {
  扫描顶点表，建立入度为零的顶点栈;
  while(栈不空) {
    弹出栈顶 v 并输出之;
    检查 v 的出边表，将其每条出边的终点 w 的入度减 1，若变为零，则 w 入栈;
  }
  若输出的顶点数小于 n，则输出“有回路”，否则拓扑排序正常结束;
}
```

下面给出求精后的拓扑排序算法：

```
int topsort(lk_graph *g) {
  lkstack S;            //假设采用链栈(结点数据类型为 int，存放入度为零的顶点号)
  pointer p;
  int m,i,v;
  init_lkstack(&S);
  for(i=1;i<=g->n;i++)
    if(g->adjlist[i].in==0) push_lkstack(&S,i);
  m=0;
  while(!empty_lkstack(&S) {
    pop_lkstack(&S,&v);cout<<v<<" ";
    m++;
    p=g->adjlist[v].first;
    while(p!=NULL) {        //将 v 点的各出边的终点 w 的入度减 1，若变为零，则入栈
      g->adjlist[p->no].in--;
      if(g->adjlist[p->no].in==0) push_lkstack(&S,p->no);
      p=p->next;
    }
  }
  if(m<g->n) {cout<<"图中有环，不能拓扑排序！\n";return 0;}
  else return 1;
}
```

分析上述算法，设有向图有 n 个顶点和 e 条边，建立初始入度为 0 的顶点栈，要检查所有顶点一次，执行时间为 O(n)；拓扑排序中，若有向图无回路，则每个顶点入、出栈各一次，每个边表结点被检查一次，执行时间是 O(n+e)。所以，总的时间复杂度为 O(n+e)。

按上述算法和图 6.26 的邻接表，对图 6.23 进行拓扑排序的过程正是图 6.25（b）～图 6.25（g）所示的过程。如刚开始时 c_1 和 c_2 的入度为 0，但 c_2 后入栈，所以先输出 c_2，删除 c_2 及其出边后 c_3 的入度为 0，c_3 入栈后栈顶为 c_3，所以它又比 c_1 先输出等。显然，邻接表不同时，拓扑序列也可能不同。

值得指出的是，上述算法的链栈可不必使用额外的空间，而将顶点表中入度为 0 的结点作为链栈的结点，具体就是将这些顶点的入度域当作链栈的 next 链指针（下标值）。这是因为入度为 0 的点找到后，其入度值就没用了，所以入度域可作它用，这里用作链指针，从而形成一个链表，起到链栈的作用。这时的链栈是一个静态链表。进一步，还可不理会链栈的结点域，因为该结点在顶点表的位置就是其当前位置，不需要由结点域来指出了，故入栈时只需修改链指针。初始化时，栈顶指针 top 置为 0，再对顶点表扫描，每找到一个初始入度为 0 的顶点，就将该点入栈，入栈操作是令其 next 指针（in 域）指向当前的栈顶（top），然后将 top 指向该顶点。这个过程逻辑上与一般的链栈是一致的，出栈亦然，具体就不细述了。

另外，为了保存当前入度为 0 的点，也可以不用栈而使用队列，它们的区别是，入度为零的点输出的时机不同：用栈时后出现的入度为 0 的点先输出，用队列时则后输出。

对上述方法适当修改，就可在求拓扑序列的过程中标识出可并行的活动（顶点）。具体做法是，初始时将所有入度为 0 的点标为一组可并行点。这组点执行完步骤（2）后，将新产生的入度为 0 的点标为新的一组可并行点，依此类推。如果对可同时进行的活动的数量有限制，如一个学期不可能学太多门的课程，则将当前并行组中多出来的点并到下一组中即可。

以上算法每次输出的是入度为 0 的点，这是按拓扑排序的本来含义进行的。反之，如

果每次输出的是出度为 0 的点，则将得到一个逆向的拓扑排序。这种算法可方便地在逆邻接表上进行：对每个结点，增加一个出度域，每次输出一个出度为 0 的顶点后，将其入边表的各顶点的出度减 1；为避免每次查找出度为 0 的点，也可用栈或队列保存当前出度为 0 的点。这些过程与上述算法类似，故具体算法从略。

逆序的拓扑排序还可在深度优先搜索遍历的基础上进行。即在深度优先搜索过程中，当访问某顶点时先不输出，而将它存入某个栈，直到从该顶点出发的所有搜索过程完成，最后回退到该点时，才将它从栈中弹出并输出。显然，最先退栈的顶点，其出度为 0，它是拓扑序列的最后一个顶点；当某个顶点 v_i 退栈时，以它为起点的边的终点都已退栈，相当于 v_i 的出度也为 0。所以这样得到的顶点序列也是一个逆向的拓扑序列。但是，不论有向图是否有环，DFS 本身都能进行遍历，所以在具体实现时要对 DFS 算法进行修改，以判断有向图是否有环，否则可能得到一个假的拓扑序列。

最后，我们应该认识到，拓扑排序也相当于是对有向图的一种遍历。

6.7.2　关键路径

对一个工程问题来说，除了关心各个子工程之间的先后关系之外，通常更关心整个工程完成的最短时间、哪些活动是影响整个工程进度的关键等问题。这就是 DAG 的另一个应用——描述工程进度的关键路径问题。在描述这类问题的有向图中，顶点表示事件，边表示活动，边上的权表示活动持续的时间，这种有向图称为**边表示活动的网**（Activity On Edge network，AOE 网）。

在 AOE 网中，顶点所表示的事件实际上就是其入边所表示的活动均已完成、其出边所表示的活动可以开始的这样一种状态。例如，图 6.27 所示的 AOE 网包括 11 项活动，9 个事件（状态）。事件 v_1 表示整个工程开始，v_9 表示整个工程结束。事件 v_5 表示活动 a_4 和 a_5 已经完成，活动 a_7 和 a_8 可以开始这种状态。若权表示的时间单位是天，则活动 a_1 需要 6 天完成，a_2 需要 4 天完成等。整个工程一开始，活动 a_1、a_2、a_3 就可并行地进行，而活动 a_4、a_5、a_6 只有当事件 v_2、v_3、v_4 分别发生后才能进行，当活动 a_{10}、a_{11} 完成时，整个工程也就完成了。

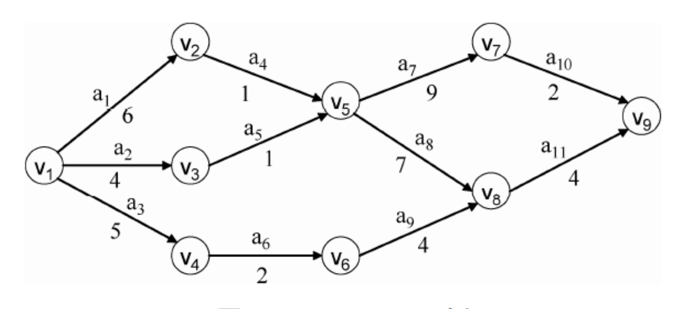


图 6.27　AOE 网示例

表示实际工程计划的 AOE 网应该是无回路的，并且网中只有一个入度为 0 的顶点（称为源点、始点，表示工程开始）和一个出度为 0 的顶点（称为汇点、终点，表示工程结束）。

1．关键路径和关键活动

AOE 网中从始点到终点的路径可能有多条，其中有些活动可以并行地进行，但显然只

有各路径上的所有活动都完成后，整个工程才算完成。所以，完成整个工程的最短时间是从源点 v_1 到终点 v_n 的最长路径的长度。这里，路径长度是路径上各边的权值之和。从源点到汇点的最长路径称为**关键路径**（Critical Path）。

显然，关键路径决定着 AOE 网的工期，关键路径的长度就是 AOE 网代表的工程所需的最短工期。图 6.27 中路径$<v_1, v_2, v_5, v_8, v_9>$就是一条关键路径，长度为 18，即整个工程至少要 18 天才能完成。一个 AOE 网的关键路径可能不止一条，如路径$<v_1, v_2, v_5, v_7, v_9>$也是图 6.27 的一条关键路径，它的长度也是 18。

为了寻找和分析关键路径，需要引入几个时间概念。

（1）一个事件 v_k 可能的最早发生时间 ve(k)，是从源点 v_1 到顶点 v_k 的最长路径长度。事件 v_k 发生后，以 v_k 为起点的各出边 $<v_k, v_y>$ 所表示的活动 a_i 才可以开始，即这些活动 a_i 的最早开始时间：

e(i)=ve(k)

例如，图 6.27 中事件 v_5 的最早发生时间是 7，则以 v_5 为起点的两条出边所表示的活动 a_7 和 a_8 的最早开始时间也是 7，即 e(7)=e(8)=ve(5)=7。

通常将源点事件 v_1 的最早发生时间定义为 0。对于事件 v_k，仅当其所有前趋事件 v_x 均已发生、且所有入边$<v_x, v_k>$表示的活动均已完成时才可能发生。所以 ve(k)可递推计算：

$$\begin{cases} ve(1) = 0 \\ ve(k) = \max\{ve(x) + w_{xk}\} \qquad <v_x, v_k> \in p[k],\ 2 \leqslant k \leqslant n \end{cases}$$

其中 p[k]表示所有以 v_k 为终点的边集， w_{xk} 表示边$<v_x, v_k>$的权。

（2）在不拖延整个工期的条件下，一个事件 v_k 允许的最迟发生时间 vl(k)，应该等于终点 v_n 的最早发生时间 ve(n)减去 v_k 到 v_n 的最长路径长度。事件 v_k 发生时，以 v_k 为终点的各入边$<v_x, v_k>$所表示的活动 a_i 均已完成，即这些活动 a_i 的最迟完成时间等于 vl(k)。由于活动 a_i 的持续时间是 w_{xk}，所以活动 a_i 的最迟开始时间：

$l(i)=vl(k)-w_{xk}$

通常将汇点事件 v_n 的最早发生时间（即工程的最早完工时间）作为 v_n 的最迟发生时间。显然事件 v_k 的最迟发生时间 vl(k)不得迟于其后继事件 v_y 的最迟发生时间 vl(y)与活动$<v_k, v_y>$的持续时间之差。所以 vl(k)可如下递推计算：

$$\begin{cases} vl(n) = ve(n) \\ vl(k) = \min\{v1(y) - w_{ky}\} \qquad <v_k, v_y> \in s[k],\ 1 \leqslant k \leqslant n-1 \end{cases}$$

其中 s[k]表示所有以 v_k 为起点的边集，w_{ky} 表示边 $<v_k, v_y>$ 的权。

（3）时间差 l(i)−e(i)表示完成活动 a_i 的时间余量，也就是在不拖延整个工期的条件下，该活动可以延迟的时间。

若时间余量为零，即 l(i)=e(i)，则称 a_i 为**关键活动**，因为它的提前或延期就会影响整个工期（提前或延期）。显然，关键路径上的活动都是关键活动。对非关键活动，它的提前完成并不能加快整个工程进度，而它的延期只要不超过其最大可利用时间，也不会影响整个工期。例如，对图 6.27 中事件 a_6，e(6)=5，l(6)=8，这意味着即使 a_6 推迟 3 天也不会延误整个工程的进度。

2. 关键路径的识别

进行关键路径分析，目的是寻找合理的资源（指能使活动进行的人力或物力）调配方案，使 AOE 网代表的工程尽快完成。为此须先识别关键路径。只有缩短关键路径上的活动（关键活动）才有可能缩短整个工期。

由前述可知，若把所有活动的最早开始时间和最迟开始时间都计算出来，就可以找出所有的关键活动，从而得到关键路径。以图 6.27 为例，各时间的计算结果如下。

（1）各事件的最早发生时间：

$ve(1)=0$
$ve(2)=ve(1)+w_{12}=0+6=6$
$ve(3)=ve(1)+w_{13}=0+4=4$
$ve(4)=ve(1)+w_{14}=0+5=5$
$ve(5)=max\{ve(2)+w_{25},ve(3)+w_{35}\}=max\{6+1,4+1\}=7$
$ve(6)=ve(4)+w_{46}=5+2=7$
$ve(7)=ve(5)+w_{57}=7+9=16$
$ve(8)=max\{ve(5)+w_{58},ve(6)+w_{68}\}=max\{7+7,7+4\}=14$
$ve(9)=max\{ve(7)+w_{79},ve(8)+w_{89}\}=max\{16+2,14+4\}=18$

不难看出，上述计算应按某一拓扑序列的次序进行。

（2）各事件的最晚发生时间：

$vl(9)=ve(9)=18$
$vl(8)=vl(9)-w_{89}=18-4=14$
$vl(7)=vl(9)-w_{79}=18-2=16$
$vl(6)=vl(8)-w_{68}=14-4=10$
$vl(5)=min\{vl(8)-w_{58},vl(7)-w_{57}\}=min\{14-7,16-9\}=7$
$vl(4)=vl(6)-w_{46}=10-2=8$
$vl(3)=vl(5)-w_{35}=7-1=6$
$vl(2)=vl(5)-w_{25}=7-1=6$
$vl(1)=min\{vl(2)-w_{12},vl(3)-w_{13},vl(4)-w_{14}\}=min\{6-6,6-4,8-5\}=0$

显然，上述计算应按某一拓扑序列的逆序进行。

根据 ve 和 vl，就可求出各活动的最早开始时间 e(i)和最迟开始时间 l(i)，以及时间余量，见表 6.5 所示。从中可见，a_1、a_4、a_7、a_8、a_{10} 和 a_{11} 是关键活动，若将图 6.27 中所有非关键活动删除，则得到图 6.28 实线所示的关键路径。

表 6.5 图 6.27 所示 AOE 网的计算结果

活动	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
e	0	0	0	6	4	5	7	7	7	16	14
l	0	2	3	6	6	8	7	7	10	16	14
l−e	0	2	3	0	2	3	0	0	3	0	0

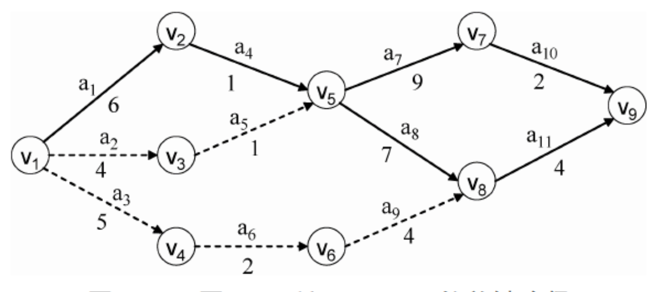


图 6.28 图 6.27 所示 AOE 网的关键路径

由前面的讨论可以得出求关键活动算法的基本步骤：

（1）对 AOE 网进行拓扑排序，以便按拓扑序列的次序求出各顶点事件的最早发生时间 ve，若图中有回路，则算法终止，否则执行步骤（2）。

（2）按拓扑序列的逆序求出各顶点事件的最迟发生时间 vl。

（3）根据各顶点的 ve 和 vl，求出各活动的最早开始时间 e(i)和最迟开始时间 l(i)。若 e(i)=l(i)，则 a_i 为关键活动。

这里第（2）步要用到逆序的拓扑序列，所以在第（1）步中要保留拓扑排序的结果。上一节的拓扑排序算法不能直接采用，因为它用栈保存入度为 0 的顶点，排序结束后栈空，拓扑序列没有保留下来。一种修改方法是在拓扑排序中输出各顶点时，用另一个栈将之保存起来，排序结束后出栈就可得到所需的逆序拓扑序列，但这要增加 O(n)的辅助空间。

如果在拓扑排序中用队列保存入度为 0 的顶点，并注意到顺序队列出队时原队列元素并没有被真正擦除（参见图 3.8，注意不是循环队列），则在拓扑排序结束后可从顺序队列的这些历史“痕迹”中得到所需的逆序拓扑序列。按此思想，求关键活动的算法如下：

```
struct node {          //边表结点
  int no;              //邻接点域
  int w;               //权(边长，活动持续时间)
  pointer next;        //链域
};
int topsort(lk_graph *g,sqqueue *S) {   //S 为顺序队列
//ve[]为顶点最早发生时间(假设为全局量)
  pointer p;
  int m,i,v;
  for(i=1;i<=g->n;i++) ve[i]=0;          //初始化顶点最早发生时间
  init_sqqueue(&S);
  for(i=1;i<=g->n;i++)
    if(g->adjlist[i].in==0) en_sqqueue(&S,i);
  m=0;
  while(!empty_sqqueue(&S)) {
    de_sqqueue(&S,&v);
    m++;
    p=g->adjlist[v].first;
    while(p!=NULL) {          //将 v 点的各出边的终点 w 的入度减 1，若变为零，则入队
      g->adjlist[p->no].in--;
      if(g->adjlist[p->no].in==0) en_sqqueue(&S,p->no);
      if(ve[v]+p->w>ve[p->no]) ve[p->no]=ve[v]+p->w;//顶点最早发生时间
      p=p->next;
    }
  }
  if(m<g->n) {cout<<"图中有环，不能拓扑排序！\n";return 0;}
  else return 1;
}
int criticalpath(lk_graph *g) {
//ve[],vl[]为顶点最早、最迟发生时间(假设为全局量)
  pointer p;
  int m,i,v,e,l;
  sqqueue S;                             //S 为顺序队列
  if(!topsort(g,&S)) {cout<<"图中有环，没有关键路径！\n";return 0;}
  for(i=1;i<=g->n;i++) vl[i]=ve[n];      //初始化顶点最迟发生时间
```

```
  for(i=g->n-1;i>=1;i--) {                //按拓扑序列逆序取顶点
    v=S[i];
    p=g->adjlist[v].first;
    while(p!=NULL) {
      if(vl[p->no]-p->w<vl[v]) vl[v]=vl[p->no]-p->w;//顶点最迟发生时间
      p=p->next;
    }
  }
  m=0;                                    //边(活动)计数器
  for(i=1;i<=g->n;i++) {                  //依次取各顶点
    p=g->adjlist[i].first;
    while(p!=NULL) {
      m++;
      e=ve[i];
      l=vl[p->no]-p->w;
      cout<<m<<" "<<g->adjlist[i].data<<" "<<g->adjlist[p->no].data<<" "
          <<e<<" "<<l<<" "<<l-e;
      if(l==e) cout<<"关键活动";
      cout<<"\n";
      p=p->next;
    }
  }
  return 1;
}
```

显然，上述算法的时间复杂度为 O(n+e)。

需要指出，缩短关键活动的时间并非一定能缩短工期；即使能缩短也不一定是等量的。因为工期由当前最长路径决定，若原关键路径缩短后还是当前的最长路径，则工期会等量地缩短；若原来有多条关键路径，其中一条缩短而其他未变，则工期不变；若出现新的最长路径，因它肯定比原来的关键路径短，故工期会缩短，但少于原关键路径的缩短量。

例如，若把图 6.28 中的关键活动 a_{11} 由 4 天缩短为 3 天，则工期并不变，因为另一条关键路径<v_1, v_2, v_5, v_7, v_9>的长度未变（还是 18 天）。若把 a_1 由 6 天缩短为 4 天，则原关键路径还是当前最长的，工期便可提前 2 天。但若把 a_1 由 6 天缩短为 3 天，则原关键路径就不是当前最长的了，新关键路径<v_1, v_3, v_5, v_7(或 v_8), v_9>的长度为 16 天，即工期只能提前 2 天。

一般地，若原来只有一条关键路径，加快其中任一活动，或者原来有多条关键路径，加快公共路径上的活动，都可缩短工期，但不一定是等量的。

习　题　六

6.1 （1）某图有 n 个顶点，则顶点的度最大可能为多少？度为奇数的点可能是奇数个吗？

（2）某无向图所有顶点的度都≥2，能否肯定它一定有回路？

6.2 （1）某无向图有 25 条边，则该图至少几个顶点？

（2）某无向图有 10 个顶点，5 条边，则该图的连通分量最多几个？最少几个？

6.3 （1）某无向图有 n 个顶点，e 条边（n>e），且是一个森林，则它有多少棵树？

（2）某无向图有 n 个顶点，m 个连通分量，则其生成森林中有几条边？

6.4 （1）某图的邻接矩阵是对角线以下为零的上三角矩阵，则该图能否进行拓扑排序？

（2）若某图可以拓扑排序，则能否对顶点重新编号使邻接矩阵是对角线以下为零的上三角矩阵？

6.5 怎样判断一个有向图是否有回路？

6.6 若对有向图经常要 DFS 遍历、BFS 遍历，求邻点、求顶点的度，则采用邻接矩阵表示和邻接表表示哪个更好？

6.7 画出图 6.29 的邻接矩阵、邻接表、逆邻接表、强连通分量。

6.8 已知某无向图有 6 个顶点，现依次输入各边(v_1, v_2)、(v_2, v_6)、(v_2, v_3)、(v_3, v_6)、(v_6, v_4)、(v_6, v_5)、(v_4, v_5)、(v_5, v_1)，采用头插法建立邻接表，试画出邻接表，并写出在此基础上从顶点 v_2 出发的 DFS 和 BFS 遍历序列。

6.9* 怎样求有向图的强连通分量？

6.10* 对图 6.30 用 Dijkstra 算法求源点 v_1 到其余各顶点的最短路径。

6.11* 对图 6.31 用 Floyd 算法求所有顶点对之间的最短路径，写出迭代过程和结果。

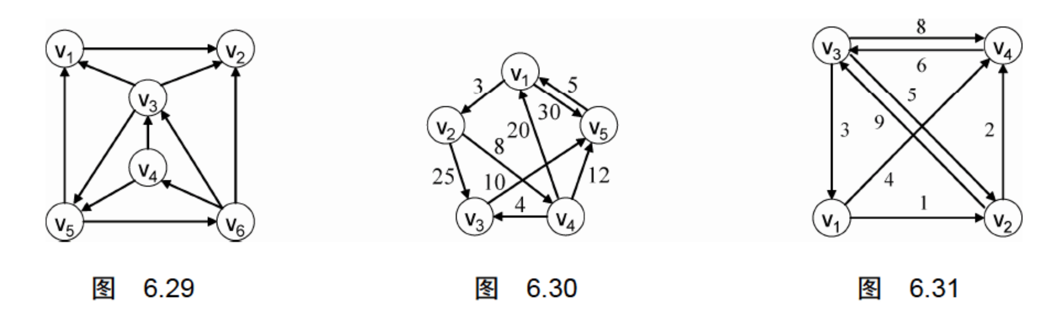


图 6.29　　图 6.30　　图 6.31

6.12* 若上题的每个顶点表示一个村庄，边上的数字表示从一个村庄到另一个所花费的时间。现要在其中之一建俱乐部，请问：

（1）若要求其他村庄到俱乐部的距离都较近，则选哪个较好？

（2）若要求俱乐部到其他村庄的距离都较近，则选哪个较好？

（3）若要求其他村庄到俱乐部的距离总和最小，则选哪个较好？

（4）若要求俱乐部到其他村庄的距离总和最小，则选哪个较好？

6.13 画出图 6.32 带入度域的邻接表，假设邻接表的结点按结点序号递增排列。分别用栈和队列保存拓扑排序中入度为零的点，写出相应的拓扑排序序列。

6.14 求图 6.33 的关键路径。

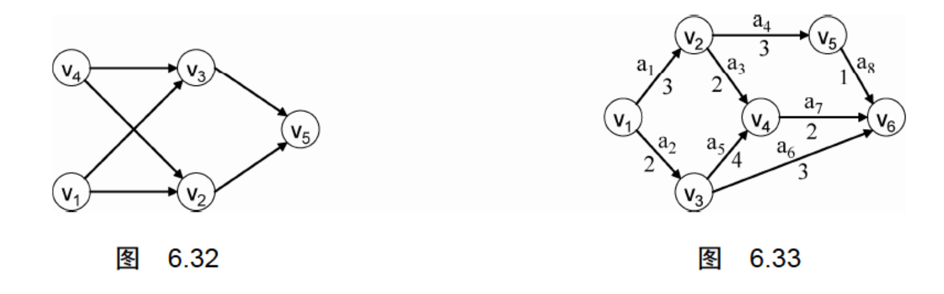


图 6.32　　图 6.33

6.15　编写算法，根据输入的顶点和边建立有向图的逆邻接表。

6.16　编写算法，由无向图的邻接矩阵生成邻接表，要求邻点表中的结点按顶点序号的大小顺序排列。

6.17　试写出深度优先遍历的非递归算法。

6.18　编写算法，实现以下功能：

（1）判断有向图中是否有从顶点 v_i 到 v_j 的简单路径（$i \neq j$），若有则输出该路径。

（2）判断有向图中是否有包含所有顶点的简单路径。

6.19　编写算法，实现以下功能：

（1）判断有向图中是否有从顶点 v 出发的简单回路，若有则输出该回路。

（2）判断有向图中是否有包含所有顶点的简单回路。

6.20　编写算法，实现以下功能：

（1）求有向图中距离顶点 v 的最短路径长度为 len 的所有顶点。

（2）求有向图中距离顶点 v 的最短路径长度最大的所有顶点。

第7章 排序

排序是数据处理中经常使用的一种重要运算。在当今的计算机系统中，花费在排序上的时间占系统 CPU 运行时间的很大比重。如何进行排序，特别是如何高效地进行排序是计算机应用中的重要课题之一，人们已经研究出了各种各样的排序方法，并且还在不断地研究和发展中。

本章介绍排序的一些基本概念和内部排序方法中几个比较经典的方法，最后对外部排序作了简单介绍。认真学习和领会各种排序方法所包含的思想和技巧对提高程序设计能力也是非常有益的。

7.1 基本概念

排序的例子在日常生活中是比较常见的，例如，电话号码簿、图书馆书目、词典、仓库清单等，一般都被整理得井井有条，这个整理的过程就是排序。排序的目的主要是为了便于以后查找，从而提高工作效率。

在本章中，我们将被排序的对象即数据元素一般称为记录。记录一般由若干个数据项（又称作域）组成，其中可用来标识一个记录的数据项或其组合，称为**关键字**（Key），简称**键**。该数据项的值称为**键值**。注意，关键字可为一个以上数据项的组合，但本章只考虑仅含一个数据项的关键字。关键字可用作排序的依据，它一般为数值型或字符串。原则上任何数据项都可作关键字，但有些关键字可唯一地标识一个记录，即不同记录该数据项的值不同，这种关键字称为**主关键字**（Primary Key）；相应地，其他不能唯一地标识一个记录的关键字称为**次关键字**（Secondary Key）。

选取记录中的哪一项作关键字，要根据问题的具体要求而定。例如，假设学生成绩表中有学号、姓名、数学、物理、化学、英语等内容，每个学生的信息是一个记录，若要唯一地标识一个记录，可用“学号”作关键字，若要按数学成绩排名次，则需用“数学”作关键字。

排序（Sorting）就是将一组任意排列的数据元素进行整理，使之重新排列成一个按关键字递增或递减的序列。或者确切地说，排序是这样的一种运算，它将含有 n 个记录的集合$\{R_1, R_2, \cdots, R_n\}$，重新排列成一个序列$\{R_{i1}, R_{i2}, \cdots, R_{in}\}$，使得相应的键值满足 $k_{i1} \leq k_{i2} \leq \cdots \leq k_{in}$（升序）或 $k_{i1} \geq k_{i2} \geq \cdots \geq k_{in}$（降序）。

注意，这里的比较运算“≤“或”≥”，不一定是数值比较，也可以是字符串比较，

甚至是用户自定义类型的比较（在 C++中可重载比较运算符）。

显然，当数据表中各记录键值均不相同时，排序的结果是唯一的，否则结果不唯一。如果多个键值相同的记录，排序后相对次序总能保持不变，则称这种排序方法是**稳定的**；否则称为不稳定的。注意，稳定性是算法本身的特性，若“稳定”则对所有情况都成立；若“不稳定”，只要举出一个反例即可。

对多关键字情况，还会涉及**多关键字排序**（多键排序）问题，即先按第一关键字排序；若第一关键字相同，则按第二关键字排序；若第二关键字也相同，再按第三关键字排序；其余类推。多关键字排序的实现一般有两种方式（设第 1 到第 d 个关键字分别为 $k_1, k_2, \cdots, k_d$）。

方法一：

依次对记录进行 d 次排序。具体实现时又有两种方式：

（1）第一次按 k_1 排序，得到若干子序列，每个子序列中 k_1 相同；然后对每个子序列按 k_2 排序，得到若干更小的子序列，再对每个子序列按 k_3 排序，……，最后对每个子序列按 k_d 排序。这种方法称为**最高位优先**（Most Significant Digit First，MSD）法。

（2）第一次按 k_d 排序，第二次按 k_{d-1} 排序……，最后按 k_1 排序。这种方法称为**最低位优先**（Least Significant Digit First，LSD）法。它每次不必将序列逐层分割成若干子序列后再分别排序，而是整体重排，比较方便。

方法二：

将关键字 $k_1, k_2, \cdots, k_d$ 分别视为字符串，将它们依次首尾连接在一起，形成一个大字符串，然后，对数据表按此字符串排序。

显然，不论哪种方法，多关键字排序问题都转化成了单关键字排序，所以，本章我们只讨论单关键字排序的情况。

根据排序过程涉及的存储设备的不同，排序可分为以下 2 种：

（1）**内排序**（Internal Sorting）。排序过程中，数据全部都存放在内存中进行处理，不涉及数据的内外存交换。

（2）**外排序**（External Sorting）。排序过程中要进行数据的内外存交换。

内排序适用于数据量小、记录个数不多的情形；外排序则针对数据量大、不能一次全部装入内存的情形，这时数据的主要部分存放在外存中，通过数据的内外存交换，借助内存逐步调整记录之间的相对位置。本章先讨论内部排序，最后简单介绍一下外部排序。

内排序的方法很多，按排序所用策略的不同，大致可分为五类：插入排序、选择排序、交换排序、归并排序和分配排序。但从排序过程中整个数据表呈现的总体变化趋势上看，这些排序方法也可分成如下两类：

（1）**有序区增长法**。将数据表分成有序区和无序区，随着排序过程的进行，有序区逐渐增长，无序区逐渐缩小，最后全部为有序区。根据具体算法的不同，初始有序区为空或仅含一个元素（一条记录总是有序的）。

（2）**有序度增长法**。数据表不能分成明显的有序区和无序区，但排序过程的每一步，整个数据表的有序程度都提高一点，最后变得完全有序。

要在众多的排序法中，简单地判断哪一种算法最好，以便能普遍选用是困难的。一般评价排序算法好坏的标准主要有两条：一是算法执行所需要的时间；二是算法执行所需要

的附加空间。另外，算法的稳定性、复杂程度等也是常考虑的因素。由于排序算法所需的附加空间一般都不大，矛盾并不突出，而排序是经常执行的一种运算，往往属于系统的核心部分，所以排序的时间开销是算法好坏的最重要的标志。

在排序过程中，一般要进行下列两种基本操作：比较关键字的大小，将记录从一个位置移动到另一个位置。所以，排序的时间开销主要是指算法执行中关键字的比较次数和记录的移动次数。当键值是字串时，比较要占用较多的时间，是影响时间复杂度的主要因素；而当记录本身的数据量很大时，为了交换记录的位置，移动记录也要占用较多的时间，是影响时间复杂度的另一个主要因素。因此，在下面讨论各种内部排序算法时，我们将主要给出各算法中记录的比较次数及移动次数。

在排序的两个基本操作中，比较操作对大多数排序方法来说都是必要的，但移动操作可通过改变数据表的存储方式来避免。数据表的存储方式常用的有 3 种：

（1）**顺序存储方式**。将数据表的各记录按其在数据表中出现的先后顺序依次存放到一组连续的内存单元中（即以一维数组作为存储结构），在排序过程中对记录本身进行物理重排，即通过比较和判定，把记录移到合适的位置。

（2）**链式存储方式**。将数据表组织成链表（动态链表或静态链表），排序过程中无须移动记录，仅需修改指针即可。通常把这类排序称为（**链**）**表排序**。

（3）**索引顺序存储方式**。数据表本身按顺序存储方式存储，但另外建立一个关键字和对应存储位置的索引表，在排序时只对索引表进行物理重排而不移动原始记录本身。它可避免一般顺序存储方式中记录的移动问题。

为了简单起见，本章以数组作为数据表的存储结构，并假设关键字是整型（实际中还可为实型、字符和字符串等）。数据表定义如下：

```
const int maxsize=100;          //数据表容量，假设为 100
typedef int datatype;
typedef struct {
  datatype key;                 //关键字域
  othertype other;              //其他域（根据实际情况设置或取消）
} rectype;                      //记录类型
typedef rectype list[maxsize+1];//数据表类型，0 号单元不用
```

由于 C/C++语言数组下标从 0 开始，而记录序号一般从 1 开始，为使二者一致，将 list 类型变量的第 0 号单元空闲，不存放记录，但可用作其他用途，如“监视哨”。

若无特别声明，本章以下均按升序讨论排序；并在无歧义的情况下，将记录键值的大小简称为记录的大小；记录键值的比较，简称为记录的比较。

7.2 插入排序

插入排序的基本思想是：每次将一个待排序的记录，按大小插入到已排好的有序区适当位置，直到全部记录插入完毕为止。这类似于玩纸牌时一边抓牌一边理牌的过程：每抓一张牌就将它插到正确的位置上。本节介绍直接插入排序和希尔排序。

7.2.1 直接插入排序

直接插入排序（Straight Insertion Sort）是一种最简单的排序方法，它的基本思想是，在排序过程中，每次都将无序区中第 1 条记录插入到有序区中适当位置，使其仍保持有序。初始时，取第 1 条记录为有序区，其他记录为无序区。随着排序过程的进行，有序区不断扩大，无序区不断缩小。最终无序区为空，有序区包含了全部记录，排序结束。

以数据表（49, 38, 49′, 91, 27, 03, 97, 49″）为例，直接插入排序过程如图 7.1 所示。其中相同的关键字 49 加撇区分；方括号表示当前的有序区；虚线表示下一趟要插入的位置。

	R[1]	R[2]	R[3]	R[4]	R[5]	R[6]	R[7]	R[8]
初始:	[49]	38	49′	91	27	03	97	49″
第1趟后:	[38	49]	49′	91	27	03	97	49″
第2趟后:	[38	49	49′]	91	27	03	97	49″
第3趟后:	[38	49	49′	91]	27	03	97	49″
第4趟后:	[27	38	49	49′	91]	03	97	49″
第5趟后:	[03	27	38	49	49′	91]	97	49″
第6趟后:	[03	27	38	49	49′	91	97]	49″
第7趟后:	[03	27	38	49	49′	49″	91	97]

图 7.1 直接插入排序示例

将无序区第一个记录 R[i]（i=2, 3, …, n）插入到有序区 R[1]～R[i−1]时，可以先在有序区中找到插入位置 k（1≤k≤i−1），然后将其后记录 R[k]～R[i−1]均后移一位，腾出位置 k 以插入 R[i]。但是，更为有效的方法是将寻找插入位置和移动记录交替进行，即从有序区的后部开始，如果该位置 j（j=i−1, i−2, …, 1）的记录大于待插记录，则直接后移一位；待插记录则插入到最后空出来的位置上。算法如下：

```
void InsertSort(list R,int n) {
  int i,j;
  for(i=2;i<=n;i++) {              //依次插入 R[2],R[3],…,R[n]
    if(R[i].key>=R[i-1].key) continue;//R[i]大于有序区最后一个记录，不需插入
    R[0]=R[i];                     //R[0]是监视哨
    j=i-1;
    do {                           //查找 R[i]的插入位置
      R[j+1]=R[j];j--;             //记录后移，继续向前搜索
    } while(R[0].key<R[j].key); //省略了条件 j>=1
    R[j+1]=R[0];                   //插入 R[i]
  }
}
```

算法中引入附加记录 R[0]有两个作用：其一是进入查找循环前，保存 R[i]的副本（记录后移时会冲掉 R[i]）；其二是在 while 循环中“监视”下标变量 j 是否越界，一旦越界（即

j<1)，R[0]自动控制 while 循环的结束（此时 j=0，循环条件 R[0].key<R[j].key 不成立)，从而可避免在 while 循环中检查 j 是否越界（即省略了循环条件"j≥1")。因此，我们把 R[0]称为"**监视哨**（Sentinel)"。这种技巧，使得测试循环条件的时间大约减少一半，如果记录数很多，节约的时间可能是相当可观的。

直接插入排序算法简单明了，它由两重循环组成：外循环表示要进行 n−1 趟插入排序，内循环表示每趟排序中关键字的比较和记录的后移。若初始数据表为升序（正序)，则每趟排序中仅需进行一次关键字的比较，总的关键字比较次数为最小值 $C_{min}=n-1$；并且每趟排序中无需后移记录，总的记录移动次数最小值为 $M_{min}=0$。

反之，若初始数据表为降序（反序)，则关键字的比较次数和记录移动次数均达到最大值。此时，对 for 循环的每一个 i 值，因当前有序区 R[1]～R[i−1]的关键字均大于待插入记录 R[i]的关键字，所以 while 循环中要进行 i 次比较才终止（终止点为监视哨 R[0])，同时有序区中所有的 i−1 个记录均后移了一个位置，再加上 while 循环前后 R[0]的两次移动，则移动记录的次数为 i−1+2。可见排序过程中总的关键字比较次数的最大值 C_{max} 和总的记录移动次数的最大值 M_{max} 分别为：

$$C_{max}=\sum_{i=2}^{n} i=\frac{(n+2)(n-1)}{2}=O(n^2)$$

$$M_{max}=\sum_{i=2}^{n}(i-1+2)=\frac{(n-1)(n+4)}{2}=O(n^2)$$

由上述分析可知，数据表的初始状态不同时，直接插入排序所耗费的时间会有很大差异。最好情况是初态为正序，此时算法的时间复杂度为 O(n)，最坏情况是初态为反序，相应的时间复杂度为 $O(n^2)$。易知算法的平均时间复杂度也是 $O(n^2)$①。当初始序列基本有序或 n 较小时，它是最佳的排序方法，但对记录数 n 较大的数据表就不适合了。

直接插入排序所需的辅助空间是一个监视哨，故空间复杂度为 O(1)。直接插入排序中记录的移动是按相邻位置顺序进行的，故是稳定的。

显然，有序区也可在数据表的尾部生成，分析方法和结论类似（略)。

上述直接插入排序算法虽然简单，但效率不高。一般可作如下改进：

（1）二分/折半插入。在有序区中寻找插入位置时，利用有序性进行二分查找（每次将当前待插入元素与有序区中点位置上的元素比较，见下一章)，这时插入第 i 个元素时最多比较 $\lceil \log_2 i \rceil$ 次就能确定插入位置，总比较次数最多为 $\sum_{i=1}^{n-1}\lceil \log_2 i \rceil \approx \log_2(n-1)!=O(n\log_2 n)$，虽不及直接插入排序的最好情况，但比其平均和最坏情况好得多，不过移动次数不变。

（2）二路插入。在数据区的两端形成高低两个有序区，每次插入到其中一个，使两者长度始终相当。这样平均插入长度就只有一个有序区时的一半，即效率可提高一倍左右。

但这些改进不是实质性的，效率仍为 $O(n^2)$。下述希尔排序能使效率提高到 $O(n^2)$以下。

① 每次对有序区 R[1]～R[i−1]查找时，平均比较 $\frac{1+(i-1)}{2}=\frac{i}{2}$ 次（见下一章顺序查找)，故总的平均比较及移动次数为 $\sum_{i=1}^{n-1}\frac{i}{2}=\frac{n(n-1)}{4}=O(n^2)$。

7.2.2　希尔排序

希尔排序（Shell Sort）又称“**缩小增量排序**”（Diminishing Increment Sort），是由 D.L.Shell 在 1959 年提出来的。它的基本方法是：将数据表分成若干组，所有相隔为某个“增量”的记录为一组，在各组内进行直接插入排序；初始时增量 d_1 较大，分组较多（每组的记录数少），以后增量逐渐减少，分组减少（每组的记录数增多），直到最后增量为 1（$d_1>d_2>\cdots>d_t=1$），所有记录为同一组，再整体进行一次直接插入排序。

下面看一个具体例子。取数据表同图 7.1，增量序列取为：5, 3, 1。

第一趟排序时，$d_1=5$，整个数据表被分成 5 组，(R_1, R_6)、(R_2, R_7)、(R_3, R_8)、(R_4)、(R_5)，其中(R_4)、(R_5)自成一组。分别对各组进行直接插入排序，结果见图 7.2 的第一趟结果。

第二趟排序时，$d_2=3$，整个数据表分成三组：(R_1, R_4, R_7)、(R_2, R_5, R_8)、(R_3, R_6)，分别对各组进行直接插入排序，得到第二趟排序结果。

最后一趟排序时，$d_3=1$，即对整个数据表做直接插入排序，其结果即为有序表。整个排序过程如图 7.2 所示。

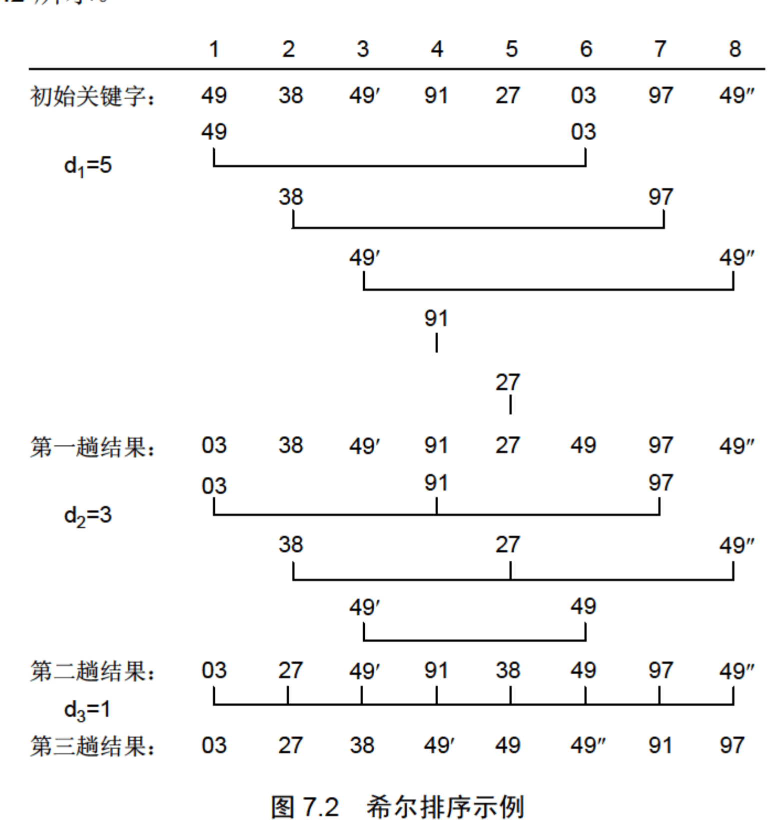


图 7.2　希尔排序示例

显然，若一开始就取增量为 1，则希尔排序就是直接插入排序。

若不设置监视哨，则每趟希尔插入排序算法如下：

```
void ShellInsert(list R,int n,int h) {//一趟插入排序，h 为本趟增量
  int i,j,k;
  for(i=1;i<=h;i++)              //i 为组号
    for(j=i+h;j<=n;j+=h) {       //每组从第 2 个记录开始插入
```

```
        if(R[j].key>=R[j-h].key) continue;//R[j]大于有序区最后一个记录，不需插入
        R[0]=R[j];                      //R[0]保存待插记录，但不是监视哨
        k=j-h;                          //待插记录的前一个记录
        do {                            //查找正确的插入位置
          R[k+h]=R[k];k=k-h;            //后移记录，继续向前搜索
        } while(k>0 && R[0].key<R[k].key);
        R[k+h]=R[0];                    //插入R[j]
      }
  }
```

注意，算法中的两个 for 循环可以合并为一个：for(j=1+h; j<=n; j++)，这相当于对每组交替着进行直接插入排序；这时循环控制的开销少了，实际执行起来也快些。

如果采用监视哨技术，则插入排序时每组都要设一个监视哨。设当前增量为 h，则各组监视哨的位置分别为 1−h, 2−h, …, 0。这些位置基本上都在数组下标范围之外，为了避免移动数据表，可在数组低端预先（按最大增量）空出若干位置。另外，为了避免对多个监视哨的赋值和每趟中的修改，可将所有监视哨都设为−∞（实取小于所有键值的数即可）。显然，如果采用尾部生成有序区的插入排序，则不必改变数据区位置，在数组高端预留足够位置即可，具体算法略（见习题 7.12）。

希尔排序过程就是调用若干趟希尔插入排序，主算法如下：

```
  void ShellSort(list R,int n){
     int h;
     for(h=第一个增量;;h=下一个增量)  //各趟插入排序
       ShellInsert(R,n,h);
       if(h==1) break;
  }
```

其中第一个增量和下一个增量的求法，取决于具体的增量序列和算法，如第一个增量=$\lfloor n/2 \rfloor$，下一个增量=max($\lfloor h/2.2 \rfloor$, 1)等。若增量序列的形成比较繁琐，也可预先求出后保存到某数组，则“第一个增量”就是在该数组中找合适的最大增量，“下一个增量”就是数组中的下一个元素。

希尔排序的效率与增量序列的选取有关，但增量序列如何选择最好，目前尚无定论。希尔的取法是 $d_1 = \lfloor n/2 \rfloor$，$d_{i+1} = \lfloor d_i/2 \rfloor$，$d_t = 1$，$t = \lfloor \log_2 n \rfloor$ ①。Knuth 建议取 $d_{i+1} = \lfloor (d_i - 1)/3 \rfloor$，$d_t = 1$，$t = \lfloor \log_3 n - 1 \rfloor$；Hibbard 取 $d_{i+1} = \lfloor (d_i - 1)/2 \rfloor$，$d_t = 1$，$t = \lfloor \log_2 n - 1 \rfloor$等。显然，增量序列的最后一个值应为 1，其他几个值之间应该没有公因子（1 除外），如 5, 3, 1。

假设增量每次除以 2，则希尔排序执行的趟数为 $O(\log_2 n)$。每趟排序时，数据分为 d_i 组，每组内数据的个数约为 n/d_i 个，所以每趟排序的效率最好为 $d_i \times O(n/d_i) = O(n)$，最坏为 $d_i \times O\left((n/d_i)^2\right) = O(n^2/d_i)$，各趟最坏的平均则为 $O(n^2/\log_2 n)$ ②。但实际上各趟分组排序

① 这时如果增量恰好为 2 的方次，如 16, 8, 4, 2, 1，则效果很差，因为前一趟比较过的数据在下一趟又位于同一组而再次比较；而后期增量为 2 时对应的 2 组数据，相互间在之前又从没有比较过。

② 不妨设 d=2^k, 2^{k-1}, …, 1，$k = \log_2 n$，则各趟最坏的平均为 $\dfrac{\sum_{i=0}^{k} \frac{n^2}{d_i}}{k+1} = \dfrac{\sum_{i=0}^{k} \frac{n^2}{2^i}}{k+1} = \dfrac{n^2 \frac{1-(1/2)^{k+1}}{1-1/2}}{k+1} \approx n^2 \dfrac{\frac{1}{1-1/2}}{k+1} = O(n^2/k) = O(n^2/\log_2 n)$。

时，数据逐渐接近有序，每趟的最坏情况随着趟数的增加基本上不会出现，所以希尔排序的整体效率在 $O(\log_2 n)\times O(n)=O(n\log_2 n)$ 和 $O(\log_2 n)\times O(n^2/\log_2 n)=O(n^2)$ 之间。但复杂性的具体表达式，除了少数特殊的增量序列外，目前还没有理论结果（但肯定与增量有关）。数值试验表明，希尔排序的平均比较次数和平均移动次数一般可达 $O(n^{1.x})$，如 $O(n^{1.25})$以下，优于直接插入排序。究其原因一般有如下两种解释：

其一，加速原理。希尔排序将相隔为某个增量的记录组成子序列，排序时子序列中某个记录向前或向后移动一位，相对于原序列而言，则移动了若干位，这有使记录加速向目标位置移动的趋势。一般地，序列的无序程度越大，记录离它的最终位置就越远，这种加速的效果就越明显。排序开始时序列的无序程度最大，而这时的增量也最大，正好适应了快速移动的要求；以后每完成某个增量下的排序后，序列的有序程度就提高了一些，相应地加速步伐就应小一些，而这时新的增量也缩小了，正好又适应了这种要求。

其二，有序和小规模原理。直接插入排序在数据表初态基本有序时时间性能较好（初态为正序时所需时间最少）；另一方面，当 n 较小时，直接插入排序的最好时间复杂度 $O(n)$ 和最坏时间复杂度 $O(n^2)$差别不大。希尔排序开始时增量较大，分组较多，每组的记录数较少，故各组内直接插入较快；后来增量 d_i 逐渐缩小，分组减少，每组的记录数增多，但由于已按增量 d_{i-1} 排过序，数据表较接近于有序状态，所以新的一趟排序也较快。

但这种分析不能解释为什么希尔排序的子序列不是简单地“逐段分割”，而是将相隔为某个增量的记录组成一个子序列这个问题。

希尔排序中记录在一定的间隔下跨越移动，可能跨过相同的关键字，从而改变它们的相对次序，故是不稳定的。

希尔排序的改进主要在增量序列的选取上，显然效果最好也只能是接近 $O(n\log_2 n)$。几个效果较好的增量序列如：

Gonnet：$h_1=\lfloor n/2\rfloor$，$h_{i+1}=\lfloor h_i/2.2\rfloor$，$h_t=1$。

Sedgewick：…, 209, 109, 41, 19, 5, 1，增量为 $9\cdot 4^i-9\cdot 2^i+1$ ($i\geqslant 0$)和 $4^j-3\cdot 2^j+1$ ($j\geqslant 2$)的合并。

以上介绍的增量序列基本上都是几何级数类型的，实验表明，其中的较好者相对于较差者改进显著，但较好者之间性能比较接近，即这类序列的改进已经越来越困难了。另外还有一类超短序列，如仅 2、3 个增量的 h, 1、h, k, 1 等，适当选取 h、k，效果也可达 $O(n^{1.x})$，如 $O(n^{5/3})$，但比几何级数类中的较好者差。具体就不细述了。

7.3 交换排序

交换排序的基本思想是：每次比较两个待排序的记录，如果发现它们的大小次序与排序要求相反时就交换两者的位置，直到没有反序的记录为止。交换排序的特点是：较大的记录向数据表的一端移动，较小的记录向数据表的另一端移动。本节介绍两种交换排序：冒泡排序和快速排序。

7.3.1 冒泡排序

冒泡排序（Bubble Sort）的基本思想是，设想数据表 R[1]～R[n]垂直放置，将每个记录 R[i]看做是重量为 R[i].key 的气泡；根据轻气泡不能在重气泡之下的原则，从下往上扫描数组 R，凡违反本原则的轻气泡，就使其向上“飘浮”，如此反复进行，直到最后任何两个气泡都是轻者在上，重者在下为止。这个过程类似气泡从水中往上冒的情形，故得名。

初始时，R[1]～R[n]为无序区。第一趟扫描从该区底部 R[n]向上依次比较相邻两个气泡的重量，若发现轻者在下，重者在上，则交换两者的位置。本趟扫描完毕时，“最轻”的气泡就飘浮到最上面，即关键字最小的记录被放在了最高位置 R[1]上。第二趟扫描时，只需扫描 R[n]～R[2]，扫描完毕时，“次轻”的气泡飘浮到 R[2]的位置上。其余依此类推。

图 7.3 是冒泡排序过程的示例，第 1 列（第 0 趟）为初始关键字，其他各列依次为各趟排序（即各趟扫描）结果，图中用方括号表示待排序的无序区。

趟次	0(初始)	1	2	3	4	5	6	7
	[49	03	03	03	03	03	03	03
	38	[49	27	27	27	27	27	27
	49′	38	[49	38	38	38	38	38
	91	49′	38	[49	49	49	49	49
	27	91	49′	49′	[49′	49′	49′	49′
	03	27	91	49″	49″	[49″	49″	49″
	97	49″	49″	91	91	91	[91	91
	49″]	97]	97]	97]	97]	97]	97]	[97]

图 7.3　上升法冒泡排序

冒泡排序也可从无序区顶部 R[1]开始向下扫描，这时每一趟是一个“最重”的气泡沉到底部。为了区分，前面的方法称为**上升法**，后面的方法称为**下沉法**。以下我们只考虑上升法。显然，对上升法，每次相邻比较后的交换中，较小者总是沿着最终位置方向进行，但另一个则可能暂时移向相反方向。

因为每一趟排序都使有序区增加一个气泡，在经过 n−1 趟排序之后，有序区中就有 n−1 个气泡，而无序区中气泡的重量总是大于等于有序区中气泡的重量，所以，整个冒泡排序过程至多需要进行 n−1 趟排序。但是，若在某一趟排序中未发现气泡位置的交换，则说明待排序的无序区中所有气泡均满足“轻者在上，重者在下”的原则。因此，冒泡排序过程可在此趟排序后终止。例如，在图 7.3 的示例中，在第四趟排序过程中就没有气泡交换位置，此时整个数据表就已达到有序状态了。为此，引入一个标志 flag，用以表示记录是否

发生过交换。在每趟排序之前，先将它置为 0，如果排序中有记录交换，则将它置为 1。这样一趟排序结束后，检查 flag，如果不为 1 则未曾有记录交换，排序结束。整个算法如下：

```
void BubbleSort(list R,int n) {     //上升法冒泡排序
  int i,j,flag;
  for(i=1;i<=n-1;i++) {           //做 n-1 趟扫描
    flag=0;                        //置未交换标志
    for(j=n;j>=i+1;j--)            //从下向上扫描
      if(R[j].key<R[j-1].key) {   //交换记录
        flag=1;
        R[0]=R[j];R[j]=R[j-1];R[j-1]=R[0];   //交换，R[0]作辅助量
      }
    if(!flag) break;               //本趟未交换过记录，排序结束
  }
}
```

容易看出，若数据表的初态是正序的，则一趟扫描就可完成排序，关键字的比较次数为 n−1，且没有记录移动。也就是说，冒泡排序在最好情况下，时间复杂度是 O(n)。

若初始数据表是反序的，则需要进行 n−1 趟排序，每趟排序要进行 n−i 次关键字的比较（1≤i≤n−1），且每次比较都必须 3 次移动记录来交换位置。这时，比较次数 C_{max} 和移动次数 M_{max} 均达到最大值：

$$C_{max}=\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}=O(n^2)$$

$$M_{max}=\sum_{i=1}^{n-1}3(n-i)=\frac{3n(n-1)}{2}=O(n^2)$$

因此，冒泡排序的最坏时间复杂度为 $O(n^2)$。

虽然冒泡排序可能在中间某趟后结束，但易知其平均排序趟数仍为 O(n)，由此可得平均比较次数仍是 $O(n^2)$，即算法的平均时间复杂度也为 $O(n^2)$。

冒泡排序需要的辅助空间为 1，用于交换记录，在上述算法中用 R[0]代替。

由于只对相邻记录进行顺序比较和交换，冒泡排序是稳定的。

上述冒泡排序算法还可作如下改进：

（1）对每趟扫描，记住最后一次发生交换的位置 last，在该位置之前的记录 R[i]～R[last−1]没有发生交换，即已经有序，所以下一趟扫描只需对 R[last]～R[n]进行。

（2）假设最重的气泡在顶部，其余气泡已有序，则对上升法需要 n−1 趟扫描（才能将该气泡沉到底部）；但对下沉法只需一趟扫描即可。反之，如果最轻的气泡在底部，其余气泡已有序，则对下沉法需要 n−1 趟扫描，对上升法只需一趟。为了改善类似情况下的效率，可以在排序过程中交替改变扫描方向，即交替使用上升法和下沉法。

这些改进也不是实质性的，效率仍为 $O(n^2)$。借助希尔排序的思想进行分组冒泡排序，也能使效率提高到 $O(n^2)$以下，但因算法是交换型的，移动次数比希尔排序多，具体略。

7.3.2 快速排序

快速排序（Quick Sort）又称划分交换排序，是霍尔（C.A.R.Hoare）1960 年发明、1962 年正式提出的。其基本思想是，在数据表中任取一个作为“基准（pivot)”，将其余记录分为两组，第一组中各记录均小于或等于基准，第二组中各记录均大于或等于基准，而基准就排在这两组中间（这也是该记录的最终位置)，这称为一趟快速排序（或一次划分)。对所分成的两组记录分别重复上述方法，直到每组只有 1 个记录为止。

快速排序的基本步骤是划分。设待划分区间为 R[p]～R[q]，不妨取无序区第 1 个记录为基准。常见的划分算法有如下三种。

划分算法 1：交替扫描（参见例 2.1)，从后向前找比基准小的记录，再从前向后找比基准大的记录，两者交换。于是将基准右边的区间 R[p+1]～R[q]分成两部分，左部分≤基准，右部分≥基准；然后将基准和左部分的最后一个交换便得到划分结果。这个算法进行中要多次交换前后两个记录，每次交换由 3 次移动实现。

划分算法 2：交替扫描，从后向前找比基准小的记录与基准交换；再从前向后找比基准大的记录与基准交换。

这相当于把算法 1 中后面小记录与前面大记录的交换改成了等效的 2 次交换：后面小记录先与基准交换、基准再与前面大记录交换。表面上看记录的移动次数更多了（2×3=6 次)，但实际上，基准在中途的各次交换都是临时的，仅最后一次交换后才是它最终的位置。于是划分中途基准的交换并不需真正进行，仅在划分完成得到最终位置后才放入该处。这样中途的等效交换中只需 2 次移动：后面小记录移到基准处、前面大记移到基准处（详见下文及图 7.4（a)），从而减少了移动次数。

划分算法 3：单向扫描，使扫描过的数据分成小值区和大等值区。对当前扫描记录，若小于基准，则把它与大等值区的首元素交换，即小值区扩大一位；否则大等值区自动扩大一位。

这个算法在进行中已形成的大值等区不是其最终位置，有一种整体上随着小值区的增长而向后移动的趋势，所以总的移动量比前面的两个算法都多。

以前述划分算法 2 为例，图 7.4（a）展示了一次划分过程，其中阴影框表示基准。具体过程为，设置两个指针 i 和 j，其初值分别为 p 和 q；将基准 R[i]保存到辅助变量 x 中。首先，令 j 从右向左扫描，直到找到 1 个小于基准 x 的记录 R[j]，将它移到位置 i 处（相当于交换 R[j]和基准 R[i]，即 x，使小于基准的记录移到基准的左边)；然后，令 i 从 i+1 起从左向右扫描，直至找到 1 个大于基准的记录 R[i]，将它移到位置 j 处（相当于交换 R[i]和基准 R[j]，即 x，使小于基准的记录移到基准的左边)；接着，再令 j 自 j−1 起向左扫描，……，如此交替改变扫描方向，从两端各自往中间靠拢，直至 i=j 时，i 便是基准 x 的最终位置，将它放在此处就完成了一次划分。

一趟划分算法如下：

```
int Partition(list R,int p,int q) {
//对无序区 R[p]~R[q]划分，返回划分后基准的位置，双向扫描
  int i,j;
  i=p;j=q;
  R[0]=R[i];          //R[0]作辅助量 x ，存放基准，基准取为无序区第一个记录
```

```
  while(i<j) {
    while(i<j && R[j].key>=R[0].key) j--;   //从右向左扫描
    if(i<j) {R[i]=R[j];i++;}                 //交换 R[i]和 R[j]
    while(i<j && R[i].key<=R[0].key) i++;   //从左向右扫描
    if(i<j) {R[j]=R[i];j--;}                 //交换 R[i]和 R[j]
  }
  R[i]=R[0];          //将基准移到最后的正确位置
  return i;
}
```

快速排序过程是一种“分治法”：通过划分得到两个子区间，对每个子区间进行同样处理后，将结果组合起来就是问题的解。这个过程是递归的，很容易用递归算法实现。设待排序区间为 R[s]～R[t]，完成一趟划分后得到基准位置 i，接着就对区间 R[s]～R[i−1]、R[i+1]～R[t]进行递归处理。主算法如下：

```
void QuickSort(list R,int s,int t) {//对 R[s]~R[t]快速排序
  int i;
  if(s>=t) return;        //只有一个记录或无记录时无需排序
  i=Partition(R,s,t);     //对 R[s]~R[t]做划分
  QuickSort(R,s,i-1);     //递归处理左区间
  QuickSort(R,i+1,t);     //递归处理右区间
}
```

对整个数据表 R 进行快速排序，只需调用 QuickSort(R, 1, n)即可。

图 7.4（b）展示了整个快速排序过程，其中方括号表示无序区。

初始关键字：	[49	38	49′	91	27	03	97	49″]
j向左扫描	↑ i					←-	←-	←- j ↑
第一次交换后	[03	38	49′	91	27	□	97	49″]
i向右扫描		↑ i -→	-→	-→		↑ j		
第二次交换后	[03	38	49′	□	27	91	97	49″]
j向左扫描，位置不变				↑ i	←- j ↑			
第三次交换后	[03	38	49′	27	□	91	97	49″]
i向右扫描					↑ ↑ i -→ j			
基准最后位置	[03	38	49′	27	49	91	97	49″]
					↑↑ i j			

（a）一次划分过程

初始关键字：	[49	38	49′	91	27	03	97	49″]
一趟排序后：	[03	38	49′	27]	49	[91	97	49″]
二趟排序后：	03	[38	49′	27]	49	[49″]	91	[97]
三趟排序后：	03	27	[38]	49′	49	49″	91	97
最后的排序结果：	03	27	38	49′	49	49″	91	97

（b）各趟排序之后的状态

图 7.4 快速排序示例

快速排序可看成冒泡排序的一种改进，比较和交换的记录前后相距较远，较大的记录一次就能交换到较后位置，较小的记录一次就能交换到较前位置。由于每次移动的距离较远，因而总的比较和移动次数较少。

快速排序过程可用一颗二叉树来描述：树根表示基准，左右子树表示划分的两个区间，每个子区间继续用子二叉树表示，这种二叉树称为快速排序的**判定树**。如图 7.4 的快速排序过程可用图 7.5 的二叉树表示。显然，树的深度就是快速排序的递归深度，由此可知递归所需栈空间的大小。最好情况下，每次划分都得到两个均匀的子序列，栈的最大深度为 $\lfloor \log_2 n \rfloor + 1$，所需栈为 $O(\log_2 n)$。最坏情况下，二叉树是一颗单枝树，递归深度为 n，所需栈空间为 O(n)。

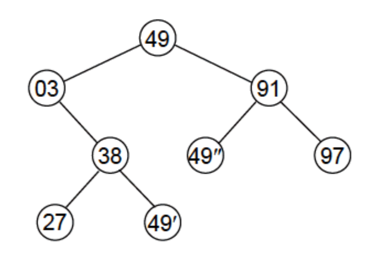


图 7.5 快速排序过程的判定树

按上述划分算法，每次都取当前无序区的第 1 个记录为基准，则快速排序的最坏情况是初始记录已经有序，这时快速排序蜕化为冒泡排序。每次划分选取的基准都是当前无序区中最小（或最大）的记录，划分的结果是基准某一侧（左边或右边）的子区间为空，另一侧子区间中的记录数仅比划分前区间的记录数少 1。因此，快速排序要做 n−1 趟，每一趟要进行 n−i 次比较，总的比较次数达到最大值：

$$C_{max} = \sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = O(n^2)$$

类似，如果初始记录已基本有序时，效率也很低。

在最好情况下，每次划分所取的基准都是当前无序区的“中值”记录，划分的结果是基准的左、右两个子区间的长度大致相等。设 C(n)表示对长度为 n 的数据表进行快速排序所需的比较次数，显然，它应该等于对长度为 n 的无序区进行划分所需的比较次数 n−1 加上递归地对划分所得的左、右两个子区间（长度≤n/2）进行快速排序所需的比较次数。假设数据表长度 $n=2^k$，则总的比较次数为：

$$\begin{aligned}
C(n) &\leqslant n + 2C(n/2) \\
&\leqslant n + 2[n/2 + 2C(n/2^2)] = 2n + 4C(n/2^2) \\
&\leqslant 2n + 4[n/4 + 2C(n/2^3)] = 3n + 8C(n/2^3) \\
&\leqslant \cdots\cdots \\
&\leqslant kn + 2^k C(n/2^k) = n\log_2 n + nC(1) \\
&= O(n\log_2 n)
\end{aligned}$$

其中 $k = \log_2 n$，C(1)表示对长度为 1 的区间进行快速排序的比较次数，为一常数。

因为快速排序的记录移动次数不大于比较的次数，所以，快速排序的最坏时间复杂度

为 $O(n^2)$，最好时间复杂度为 $O(n\log_2 n)$。

可以证明，快速排序的平均时间复杂度约为 $1.39n\log_2 n$（见习题 7.10），即仍为 $O(n\log_2 n)$，它是目前基于比较的内部排序方法中速度最快的[①]，故得名（与其名字相称）。

快速排序中，记录在前后两侧跨越移动（或交换），可能跨过相同的关键字，从而改变它们的相对位置，故是不稳定的。

对上述快速排序算法还可做很多改进，比如：

（1）为了改善最坏情况下的时间性能，可采用“三者取中”的规则，即在每一趟划分开始前，首先比较 R[p].key、R[q].key 和 R[$\lfloor (p+q)/2 \rfloor$].key，令三者中取中值的记录作为基准（与第一个交换后即可采用上述方法）。另一种常用方法是在区间中随机地选择一个作基准，一般不会出现最坏情况。

（2）为了改善递归算法的时空性能，可将上述快速排序算法改成非递归的，这需要引进一个栈（或队列），最多不超过 n，具体算法略（见习题 7.17）。

（3）因为快速排序适合 n 较大的情形，故对长度较小的子序列不必用快速排序，而用插入排序。也可对长度小的子序列什么也不做，最后得到一个没有完全排序但已有较大（分块）有序程度的序列，再对其一次性地使用插入排序。

（4）当数据表中有大量重复（等值）数据时，快速排序效率很低。这时可在每次划分完成后检查基准的前后两侧，排除等值区后再递归处理前后两部分。

7.4　选择排序

选择排序（Selection Sort）的基本思想是：每一趟从待排序的记录中选出最小（或最大）的记录，顺序放在已排好序的子序列的最后（或最前），直到全部记录排序完毕。本节介绍两种选择排序方法：直接选择排序和堆排序。

7.4.1　直接选择排序

直接选择排序（Straight Selection Sort）是一种比较简单的排序方法，它的做法是：首先，所有记录组成初始无序区 R[1]～R[n]，从中选出最小的记录，与无序区第一个记录 R[1]交换；新的无序区为 R[2]～R[n]，从中再选出最小的记录，与无序区第一个记录 R[2]交换；类似，第 i 趟排序时 R[1]～R[i−1]是有序区，无序区为 R[i]～R[n]，从中选出最小的记录，将它与无序区第一个记录 R[i]交换，R[1]～R[i]变为新的有序区。因为每趟排序都使有序区中增加一个记录，所以，进行 n−1 趟排序后，整个数据表就全部有序了。

直接选择排序的过程如图 7.6 所示，图中方括号表示当前无序区，虚线表示下一趟要交换的记录。

上述过程也可看成一种冒泡排序：每次比较和交换的是无序区的最小记录和无序区的第一个记录，而不总是相邻的两个记录。这样总的比较次数相同，但移动次数则大大减少。

① 关键字比较次数并非最少，但综合关键字移动次数和程序的其他开销，实际运行时间通常最少。

直接选择排序算法如下：

```
void SelectSort(list R,int n) {
  int i,j,k;
  for(i=1;i<=n-1;i++) {                  //n-1 趟排序
    k=i;
    for(j=i+1;j<=n;j++)                  //在当前无序区中找最小的记录 R[k]
      if(R[j].key<R[k].key) k=j;
    if(k!=i) {R[0]=R[i];R[i]=R[k];R[k]=R[0];}//交换 R[i]和 R[k]，R[0]作辅助
  }
}
```

	1	2	3	4	5	6	7	8
初始关键字：	[49	38	49′	91	27	03	97	49″]
第一趟排序后：	03	[38	49′	91	27	49	97	49″]
第二趟排序后：	03	27	[49′	91	38	49	97	49″]
第三趟排序后：	03	27	38	[91	49′	49	97	49″]
第四趟排序后：	03	27	38	49′	[91	49	97	49″]
第五趟排序后：	03	27	38	49′	49	[91	97	49″]
第六趟排序后：	03	27	38	49′	49	49″	[97	91]
第七趟排序后：	03	27	38	49′	49	49″	91	[97]
最后排序结果：	03	27	38	49′	49	49″	91	97

图 7.6　直接选择排序示例

显然，无论数据表初始状态如何，在第 i 趟排序中选出最小关键字的记录，都需做 n−i 次比较，因此，总的比较次数为：

$$C_{max}=\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}=O(n^2)$$

至于记录移动次数，当初始数据表为正序时，移动次数为 0；当数据表初态为反序时，每趟排序均要执行交换操作，所以，总的移动次数取最大值 3(n−1)。直接选择排序的平均时间复杂度为 $O(n^2)$。由于记录的移动次数较少，所以当记录本身的数据量较大时，直接选择排序比较有利。

直接选择排序的辅助空间为 1，用于交换记录，上面算法中用 R[0]代替。

直接选择排序中有记录的跨越交换（将无序区最小或最大记录与无序区第一个记录的交换），可能使键值相同记录的相对位置发生交错，故是不稳定的。

显然，有序区也可在数据表的尾部生成，分析方法和结论类似（略）。

直接选择排序的一个简单改进是二路选择：每次扫描选出最小、最大两个记录，分别与无序区首、尾记录交换，即在数据区的前后两端形成两个有序区。其中选择算法可参见习题 1.9 的代码调整，可一定程度减小比较次数，但复杂性仍是 $O(n^2)$。

7.4.2 堆排序

堆排序是对直接选择排序的一种改进。在讨论堆排序之前先介绍一下**树形选择排序**（Tree Selection Sort）。

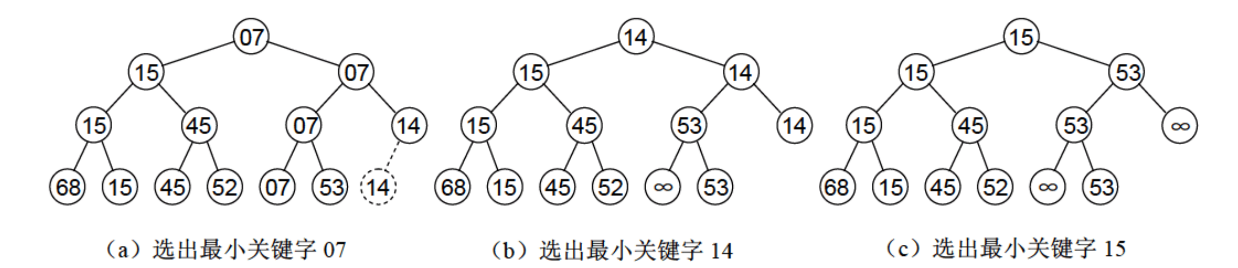
在直接选择排序中，为了从 n 个记录中找出最小的，需要进行 n−1 次比较；然后在剩下的 n−1 个记录中找次小的，又需进行 n−2 次比较，依此类推。实际上，除第一次的 n−1 次比较外，后面各次比较中有很多可能在前面已经做过，由于这些结果没有保留下来，所以在以后又重复进行。

树形选择排序可克服这一缺点，其基本思想是：首先，n 个记录每两个一组，比较后取出较小者。如果某组只有一个记录，则轮空，直接进入下一轮比较。这样得到 $\lceil n/2 \rceil$ 个较小者，然后再两两比较。如此重复，直至选出最小者为止。这个过程跟日常很多比赛一样，两两决胜负，最后决出冠军，所以有时也形象地称为**锦标赛排序**（Tournament Sort）。

上述过程可用一棵完全二叉树来表示：最底层和倒数第 2 层的叶子代表待排序的 n 个记录的键值；叶子上面一层是叶子两两比较后较小的结果；依此类推，最后树根表示选择出来的最小关键字。

将最小记录输出后，便完成了第一趟选择。然后在剩下的叶结点中，可按同样方法进行第二趟选择，得到新的最小关键字。注意到树中记录着以前比较的结果信息，所以在第二趟选择前，为了利用已有结果以及不破坏已有的树结构，可将前一趟找到的最小叶子结点的键值改为+∞，这样重新比较时，实际只需修改从树根到刚成为+∞的叶子结点这条路径上各结点的值，其他结点保持不变。由于二叉树的深度为 $\lceil \log_2 n \rceil + 1$，所以最多只需比较 $\lceil \log_2 n \rceil$ 次，而不是 n−2 次了。依次类推，经过 n−1 趟选择，就将记录按升序输出了。

图 7.7 给出了对关键字{68, 15, 45, 52, 07, 53, 14}进行树形选择排序的部分过程。



（a）选出最小关键字 07　（b）选出最小关键字 14　（c）选出最小关键字 15

图 7.7　树形选择排序的部分过程

注意，已知叶子数 n 的完全二叉树有两棵，结点总数分别是 2n−1 和 2n，其中最底层的叶子数分别为偶数和奇数，但后一种在两两比较时有一个叶子轮空直接进入上一层（如图 7.7（a）的结点 14），故只需考虑前一种情况。这时内部结点数为 n−1，即需 n−1 场比赛。

可见，除第一次需进行 n−1 次比较外，以后每次都最多经过 $\lceil \log_2 n \rceil$ 次比较就可选择出最小的键值，总的比较次数不超过 $(n-1)+(n-1)\lceil \log_2 n \rceil = O(n\log_2 n)$。由于移动次数不超过比较次数，所以树形选择排序总的时间复杂度为 $O(n\log_2 n)$。

这种方法虽然减少了比较次数，但对 n 个记录需要 2n−1 个存储单元，即增加了 n−1

个结点用于保存前面比较的结果，并且排序的结果也需要另外存储。另外，与+∞的比较实际上是多余的。

为了克服树形选择排序的缺点，威洛姆斯（J.Willioms）和弗洛伊德（Floyd）在 1964 年提出了一种改进方法，称为**堆排序**（Heap Sort），它是一种巧妙的树形选择排序，不需要专门设立树。其特点是，将 R[1]～R[n]看成是一棵完全二叉树的顺序存储结构，在排序过程中利用完全二叉树双亲和孩子之间的内在关系来选择最小（或最大）的记录。

首先，引入堆的定义。**堆**是一棵完全二叉树，其中任一非叶子结点的关键字均小于等于（或大于等于）其孩子结点的关键字①。另一种常见的定义是：n 个关键字序列 $K_1, K_2, \cdots, K_n$ 称为堆，当且仅当该序列满足：

$$K_i \leqslant K_{2i} \text{ 且 } K_i \leqslant K_{2i+1} \quad (1 \leqslant i \leqslant \lfloor n/2 \rfloor)$$

或者

$$K_i \geqslant K_{2i} \text{ 且 } K_i \geqslant K_{2i+1} \quad (1 \leqslant i \leqslant \lfloor n/2 \rfloor)$$

这两个定义本质上是一致的，但从堆的来源和使用上看，堆的树形结构定义较好。显然，堆中根结点（称为堆顶）的关键字最小（或最大），我们称之为**小根堆**（或**大根堆**）。

易知，有序表为堆，但堆不一定为有序表。由于堆中双亲和孩子间有一定的有序性，但整体不一定有序，这种有序性有时称为**“堆有序”**。

例如，关键字序列“10, 15, 56, 25, 30, 70”就是一个小根堆，它所对应的完全二叉树如图 7.8（a）所示，而图 7.8（b）是一个大根堆。显然，在堆中任一棵子树也是堆。

堆排序正是利用小根堆（或大根堆）来选取当前无序区中最小（或最大）的记录来实现排序的。我们不妨利用大根堆来排序。首先，将初始无序区调整为一个大根堆，输出最大的堆顶记录后，将剩下的 n−1 个记录再重建为堆，于是便得到次大者。如此反复执行，直到全部元素输出完，从而得到一个有序序列。这个过程就是堆排序。

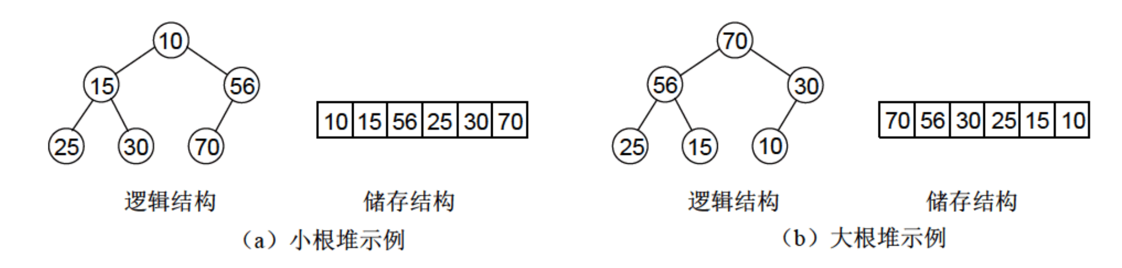


图 7.8 堆示例

为了保证时间性能，就要充分利用已有结果，即在每次输出堆顶元素后，剩下的元素不应该完全重新建堆，而应该在原堆上通过某些调整得到；为了保证空间性能，输出的堆顶也应该尽量利用原有空间，不难发现可将它与无序区中最后一个记录交换位置。这样，整个排序过程中有序区是在原记录区的尾部逐步形成并向前扩大到整个记录区的。这里有两个问题需要解决：

（1）最初时如何由一个无序序列建成一个堆？

（2）在输出堆顶元素后，如何调整剩余元素成为一个新的堆？

① 内存中有块动态储存区也叫“堆”（用于管理无后进先出特点的数据），与堆排序的堆毫无关系。

先看初始堆的建立，即把整个记录数组 R[1]～R[n]调整为一个大根堆。这要求把完全二叉树中以每一结点为根的子树都调整为堆。显然只有一个结点的树是堆，而在完全二叉树中，所有序号 $i>\lfloor n/2 \rfloor$ 的结点都是叶子，因此，以这些结点作为根的子树均已是堆。这样，我们只须依次将以序号为 $\lfloor n/2 \rfloor$，$\lfloor n/2 \rfloor-1$，…，1 的结点作为根的子树都调整为堆即可。按该次序调整每个结点时，其左、右子树均已是堆（不妨将空树亦看做是堆）。

于是问题变为，已知结点 R[i]的左、右子树已是堆，如何将以 R[i]为根的完全二叉树调整为堆？解决这一问题可采用**“筛选法”**。

筛选法的基本思想是：因为 R[i]的左、右子树已是堆，这两棵子树的根分别是各自子树中关键字最大的结点，所以，我们必须在 R[i]和它的左、右孩子中选取关键字最大的结点放到 R[i]的位置上。若 R[i]的关键字已是三者中的最大者，则无须做任何调整，以 R[i]为根的子树已构成堆；否则，必须将 R[i]和具有最大关键字的左孩子 R[2i]或右孩子 R[2i+1]进行交换。不妨设 R[2i]的关键字最大，将 R[i]和 R[2i]交换位置，交换之后有可能导致以 R[2i]为根的子树不再是堆，但由于 R[2i]的左、右子树仍然是堆，于是可重复上述过程，将以 R[2i]为根的子树调整为堆，……，如此逐层递推下去，最多可能一直调整到树叶。这一过程就像过筛子一样，把较小的关键字筛下去，而将最大关键字一层层地选择上来。

图 7.9 表示了对关键字序列：49, 38, 49′, 91, 27, 03, 97, 49″，在建堆过程中完全二叉树的变化情况，其中 n=8，故从第 4 个结点开始进行调整。

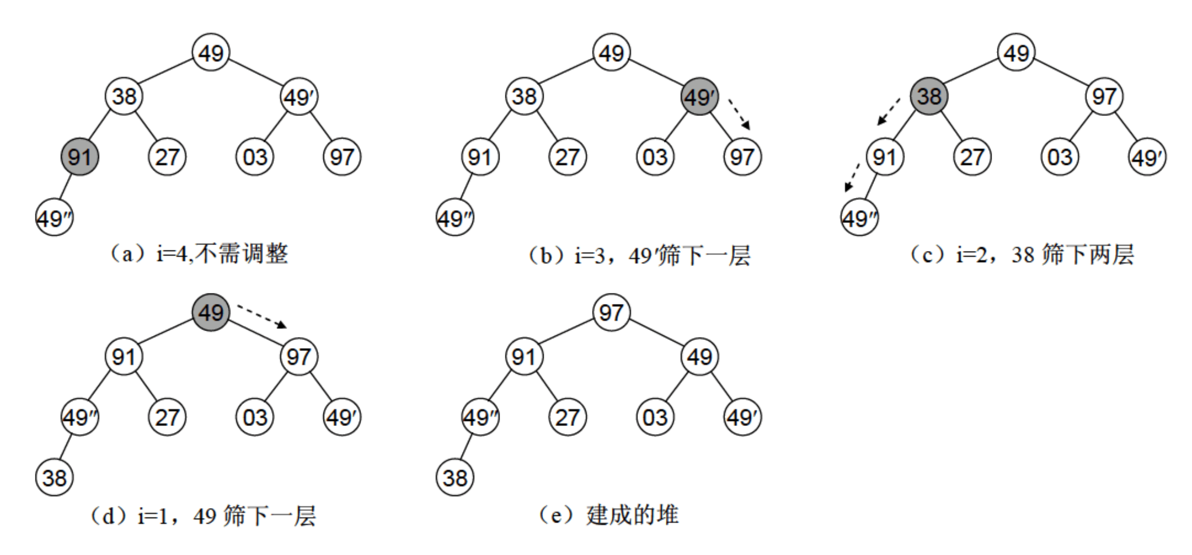


图 7.9　建堆过程示例

筛选算法如下：

```
void Sift(list R,int p,int q) {//堆范围为R[p]~R[q]，调整R[p]为堆，非递归算法
  int i,j;
  R[0]=R[p];                    //R[0]作辅助量，保存原根结点
  i=p;                          //i 指向待调整点
  j=2*i;                        //j 指向 R[i]的左孩子
  while(j<=q) {
    if(j<q && R[j].key<R[j+1].key) j++;//j 指向 R[i]的右孩子
    if(R[0].key>=R[j].key) break;      //根结点大于孩子，已经是堆，调整结束
    R[i]=R[j];                  //将 R[j]换到双亲位置上
    i=j;                        //修改当前被调整结点
    j=2*i;                      //j 指向 R[i]的左孩子
  }
```

```
  R[i]=R[0];                    //原根结点放入正确位置
}
```

其中，被调整结点 R[i]和它的某个孩子 R[j]交换时，仅将 R[j]放入了 R[i]的位置，而原 R[i]并没有放入到 R[j]的位置，这是因为它在该位置还可能会被继续筛选下去，为了减少记录的移动次数，只有当整个筛选过程结束时，才将它放到最终位置（这点类似前述快速排序的划分算法）。

另外，在完全二叉树中，若一个结点没有左孩子，则该结点必是叶子，因此上面算法中的条件 j≤q，即 2i≤q 不成立时，表示当前调整结点 R[i]已是叶子，故筛选过程结束。

由于筛选过程的递归性，也容易写出相应的递归算法，但这时待调整结点 R[i]在中途交换中每次都要真正移动，故移动次数比上述非递归形式多，具体算法略。

建堆后 R[1]是关键字最大的记录，而排序后它应该是记录区 R[1]～R[n]的最后一个记录，因此，将 R[1]和 R[n]交换后便得到了第一趟排序的结果。

第二趟排序时，首先将当前无序区 R[1]～R[n−1]调整为堆。因为第一趟排序后，R[1]～R[n−1]中只有 R[1]的值发生了变化，它的左、右子树仍然是堆，所以，这个调整过程可以调用筛选算法 Sift(R, 1, n−1)。然后，将堆顶记录 R[1]和当前无序区的最后一个记录 R[n−1]交换，结果 R[1]～R[n−2]变为新的无序区，R[n−1]～R[n]为有序区。如此重复 n−1 趟排序之后，就使有序区扩充到整个记录区 R[1]～R[n]。

图 7.10 是堆排序的全过程示例，其中虚线下的结点表示已排好序的记录，最后一步即子图(n)时，堆中只有一个结点，排序结束。

最后，堆排序算法如下：

```
void HeapSort(list R,int n) {  //对 R[1]~R[n]进行堆排序
  int i;
  for(i=n/2;i>=1;i--) Sift(R,i,n); //建初始堆
  for(i=n;i>=2;i--) {              //进行 n-1 趟堆排序
    R[0]=R[1];R[1]=R[i];R[i]=R[0]; //堆顶和当前堆底交换，R[0]作辅助量
    Sift(R,1,i-1);                 //将 R[1]~R[i-1]重建堆
  }
}
```

堆排序的时间，主要由建初始堆和不断重建堆这两部分的时间开销构成。建初始堆时从下往上分别把各层结点对应的子树调用 Sift 过程调整为堆。当堆为满二叉树时，设深度为 h（结点数 $n=2^h-1$），第 i 层结点（有 2^{i-1} 个）最多下调到第 h 层，下调 h–i 次，在 Sift 过程中关键字比较次数至多为下调次数的 2 倍，故关键字比较次数最多为

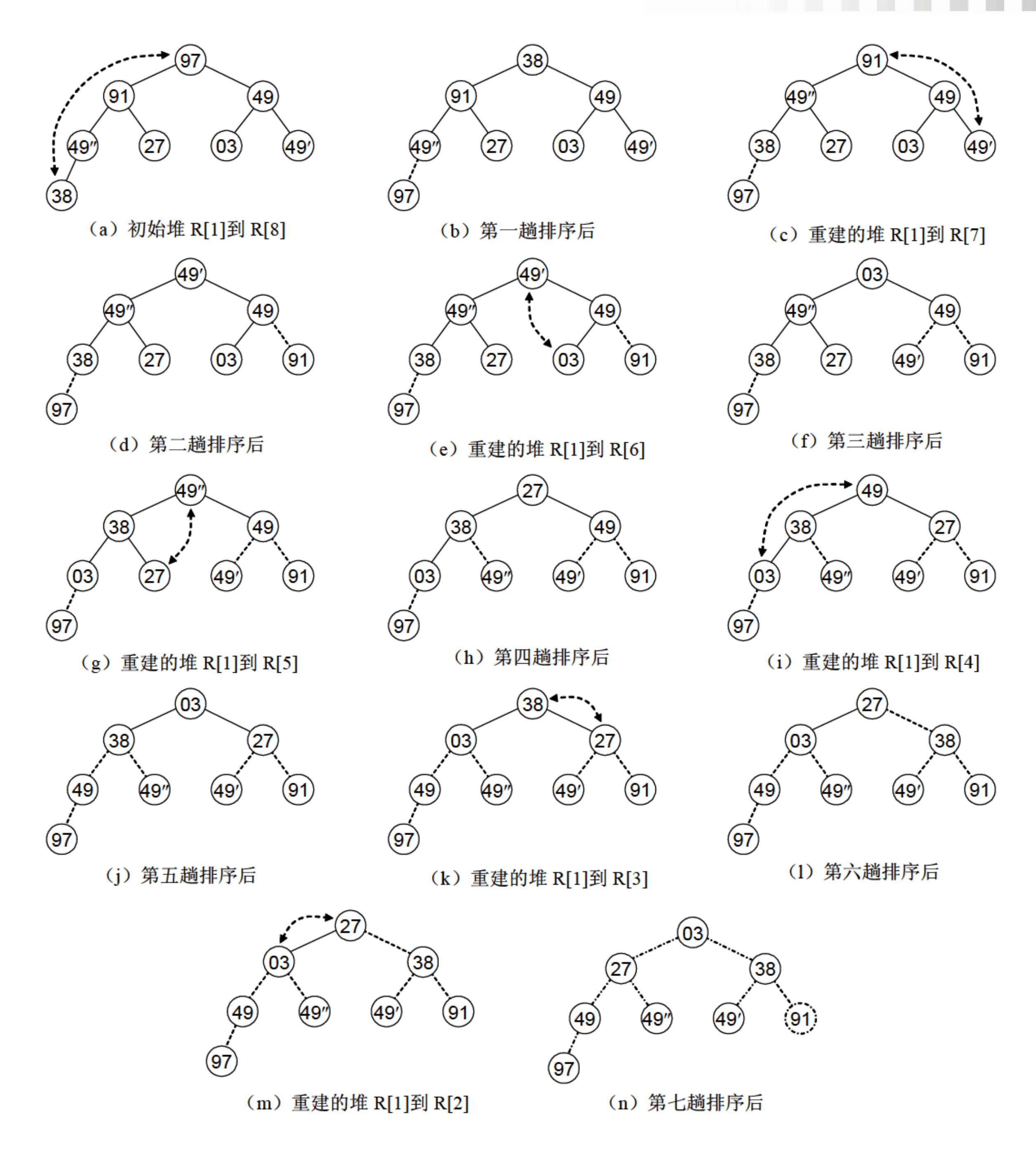
$$
\begin{aligned}
C_1'(n) &= 2\sum_{i=h-1}^{1} 2^{i-1}\bullet(h-i) = 2\left[2^{h-2}\times 1 + 2^{h-3}\times 2 + \cdots 2^0\times(h-1)\right] \\
&= 2\left[2^h-(h+1)\right] \\
&= 2\left[(n+1)-(h+1)\right] = 2\left[n-h\right] < 2n
\end{aligned}
$$

当堆不是满二叉树时，相当于在上述基础上 h +1 层的左侧增加 x 个叶子（这时 $n=2^h-1+x$），则它们的祖先结点对应的子树高度增 1，相应的下调次数最多增加 1 次。可证这些祖先数 $\Delta < x+h$，所以总的关键字比较次数不超过

$$C_1(n) = C_1'(n) + 2\Delta < 2\left[2^h-(h+1)\right] + 2\left[(x+h)\right] = 2\left[(n+1-x-h-1)+(x+h)\right] = 2n$$

可见不论堆形态如何，建堆时关键字比较次数都不超过 2n，其复杂性为 O(n)。

另可证明，建堆时平均比较次数约为 1.88n（很接近最坏情况）。



（a）初始堆 R[1]到 R[8]　（b）第一趟排序后　（c）重建的堆 R[1]到 R[7]

（d）第二趟排序后　（e）重建的堆 R[1]到 R[6]　（f）第三趟排序后

（g）重建的堆 R[1]到 R[5]　（h）第四趟排序后　（i）重建的堆 R[1]到 R[4]

（j）第五趟排序后　（k）重建的堆 R[1]到 R[3]　（l）第六趟排序后

（m）重建的堆 R[1]到 R[2]　（n）第七趟排序后

图 7.10　堆排序过程示例

第 j 次重建堆时，堆中有 n−j 个结点，完全二叉树深度为$\lfloor \log_2(n-j) \rfloor + 1$，调用 Sift 重建堆所需的比较次数至多为$2 \times \lfloor \log_2(n-j) \rfloor$。因此，n−1 趟排序过程中重建堆的比较总次数不超过 $C_2(n)$，且

$$
\begin{aligned}
C_2(n) &= \sum_{j=1}^{n-1} 2\lfloor \log_2(n-j) \rfloor \\
&\leqslant 2 \times [\log_2(n-1) + \log_2(n-2) + \cdots + \log_2 2 + \log_2 1] \\
&= 2\log_2(n-1)! \\
&\approx 2[(n-1)\log_2(n-1) - 1.5(n-1)] \\
&\approx 2n\log_2 n - 3n = O(n\log_2 n)
\end{aligned}
$$

在上述 Sift 算法中，记录的移动次数不会超过比较次数，因此，堆排序的最坏时间复杂度是 $O(n)+O(n\log_2 n)=O(n\log_2 n)$。堆排序的平均性能分析较难，但结果表明它较接近于最坏性能，即 $O(n\log_2 n)$。

堆排序需要的辅助空间为 1，用于交换记录，在上述算法中用 R[0]代替。

堆排序中，记录有跨越交换（筛选以及将堆顶与无序区最后位置交换），可能改变键值相同记录的相对位置，故是不稳定的（但锦标赛排序是稳定的）。

上述堆排序算法还可改进。比如，重建堆时因堆底相对较小（或较大），把它交换到堆顶，显得“提升”过多，以后必然“降级”较多，产生较多的比较和交换次数。对此着手改进能使算法的最坏复杂性从 $2n\log_2 n$ 左右降低到 $n\log_2 n$ 左右（平均性能接近最坏性能），具体算法略。

另外，建堆过程也可以在已有堆尾部不断添加（插入）新结点，将新结点与双亲、双亲的双亲，直到根进行比较来调整其位置（见习题 7.19）。该过程比较次数可能较多，但这种从下到上调整的思想在树形选择排序（包括后面外排序中的败者树）中经常用到。前述 Sift 过程则是从上到下的调整过程。

不难想象，把堆排序的二叉树推广到多叉树，算法也是可行的，这时比较次数在增加，而移动次数在减少，有研究表明，4～6 叉树较好，总的时间可以节省 20%～30%左右。

7.5 归并排序

归并排序（Merging Sort）是利用“归并”技术来进行排序，所谓归并是指将若干个已排序的子表合并成一个有序表。

最简单的归并是将两个有序的子表合并成一个有序表。假设 R[low]～R[mid]和 R[mid+1]～R[high]是存储在同一个数组中且相邻的两个有序的子表，要将它们合并为一个有序表 R_1[low]～R_1[high]，只要设置 3 个指示器 i、j 和 k，其初值分别是这 3 个记录区的起始位置。合并时依次比较 R[i]和 R[j]的关键字，取关键字较小的记录复制到 R_1[k]中，然后，将指向被复制记录的指示器和指向复制位置的指示器 k 分别加 1，重复这一过程，直至全部记录被复制入 R_1[low]～R_1[high]中为止。其算法如下：

```
void Merge(list R,list R1,int low,int mid,int high) {
//合并 R 的两个子表：R[low]~R[mid]、R[mid+1]~R[high]，结果在 R1 中
  int i,j,k;
  i=low;
  j=mid+1;
  k=low;
  while(i<=mid && j<=high)
    if(R[i].key<=R[j].key) R1[k++]=R[i++];  //取小者复制
    else R1[k++]=R[j++];
  while(i<=mid) R1[k++]=R[i++];              //复制左子表的剩余记录
  while(j<=high) R1[k++]=R[j++];             //复制右子表的剩余记录
}
```

归并排序就是利用上述归并操作实现排序的，其基本思想是：开始时，将数据表 R[1]～R[n]看成 n 个长度为 1 的有序子表，把这些子表两两归并，便得到 $\lceil n/2\rceil$ 个有序的子表（当

n 为奇数时，归并后仍有一个长度为 1 的子表）；然后，再把这⌈n/2⌉个有序子表两两归并，如此反复，直到最后得到一个长度为 n 的有序表为止。上述归并操作，每次都是将两个子表合并成一个子表，这种方法称为“**二路归并排序**”。类似地也可以有“三路归并排序”或“多路归并排序”。二路归并排序的全过程如图 7.11 所示，其中方括号表示有序子表。

初始关键字	49	38	49′	91	27	03	97	49″
初始子表	[49]	[38]	[49′]	[91]	[27]	[03]	[97]	[49″]
第一趟归并后	[38	49]	[49′	91]	[03	27]	[49″	97]
第二趟归并后	[38	49	49′	91]	[03	27	49″	97]
第三趟归并后	[03	27	38	49	49′	49″	91	97]

图 7.11　二路归并排序示例

在给出二路归并排序算法之前，必须先解决一趟归并问题。在一趟归并中，设各子表长度为 len（最后一个子表长度可能小于 len），则归并前 R[1]～R[n]共有⌈n/len⌉个有序的子表：R[1]～R[len]、R[len+1]～R[2×len]、…、R[(⌈n/len⌉-1)×len+1]～R[n]，调用归并操作 Merge 将相邻的一对子表进行归并时，可能最后单独剩一个子表（子表个数为奇数）、或最后两个子表中后一个长度小于 len，这两种情况要特殊处理。一趟归并算法如下：

```
void MergePass(list R,list R1,int n,int len) {//对 R 做一趟归并，结果在 R1 中
  int i,j;
  i=1;                         //i 指向第一对子表的起始点
  while(i+2*len-1<=n) {        //归并长度为 len 的两个子表
    Merge(R,R1,i,i+len-1,i+2*len-1);
    i=i+2*len;                 //i 指向下一对子表起始点
  }
  if(i+len-1<n)                //最后剩两个子表，后一个长度小于 len
    Merge(R,R1,i,i+len-1,n);
  else                         //最后剩一个子表（子表个数为奇数）
    for(j=i;j<=n;j++)          //将最后一个子表复制到 R1 中
      R1[j]=R[j];
}
```

二路归并排序就是反复调用一趟归并，将数据表进行若干趟归并，每趟归并后有序子表的长度 len 扩大一倍。第一趟归并时，有序子表的长度 len 为 1，当有序段长度 len≥n 时排序完成。二路归并算法如下：

```
void MergeSort(list R,list R1,int n) {//对 R 二路归并排序，结果在 R 中(非递归算法)
  int len;
  len=1;
  while(len<n) {
    MergePass(R,R1,n,len);len=len*2;     //奇趟归并，结果在 R1 中
    MergePass(R1,R,n,len);len=len*2;     //偶趟归并，结果在 R 中
  }
}
```

其中分奇趟和偶趟归并是为了交替使用 R1 和 R 作辅助空间。本来偶趟归并时要判断是否 len≥n，若是则排序实际已完成，但结果在 R1 中，这时再执行偶趟归并时，因 MergePass 函数中 while 和 if 条件都不满足，结果只执行了最后面的复制语句，正好将 R1 中的数据

复制到 R 中。

显然，第 i 趟归并后，有序子表长度为 2^i，因此，对于具有 n 个记录的数据表，必须做 $\lceil \log_2 n \rceil$ 趟归并。每趟归并时，每比较一次就有一个记录移动，但可能某个子区间先移动完，这时另一个子区间的剩余记录就不必比较了，即记录比较次数不超过移动次数，而记录移动次数为 n，从而每趟归并所花的时间是 O(n)。所以，二路归并排序算法的时间复杂度最好和最坏都为 $O(n\log_2 n)$。

算法中的辅助空间为数组 R1，所需的空间是 O(n)，大于前面介绍的其他排序方法（但比较次数少）。

二路归并中，若两个有序子表中存在键值相同的记录，则前一个子表中的记录先复制（若有多个则依次复制），即不会改变这些记录的相对位置，故是稳定的。

需要指出的是，二路归并实际上并不需要从单个记录开始进行两两归并，通常可以先利用直接插入排序求得较长的有序子表，然后再两两归并。因为直接插入排序是稳定的，这种改进后的归并排序仍然是稳定的。

以上算法是自底向上进行的，区间不断合并而增大。二路归并排序也可自顶向下进行，区间不断分割而变小：先分成两部分 R[1]～R[mid]、R[mid+1]～R[high]，分别排序得两个有序区间，再将两者归并。而前后两部分的排序用同样的方法：各自又分成前后两部分，分别排序，再归并。这是一种“分治法”，很容易用递归实现（与快速排序相比，这里“分”容易，而“合”较难），其时间复杂度与非递归算法相当。但递归时不能交替使用 R1 和 R 作辅助空间，Merge 归并后的结果仍要在原 R 中，这需要在归并前将数据从 R 复制到 R1 中（或归并后将数据从 R1 复制到 R 中），再加上递归栈引起的附加时空开销，使得递归算法的实际执行效率常不及非递归算法（但算法简洁些），具体略（见习题 7.18）。

7.6 分配排序

前面所讨论的排序算法都是基于关键字之间的比较，通过比较判断出谁大谁小，然后进行调整。分配排序则不然，它是利用关键字的结构，通过“分配”和“收集”的办法来实现排序的，排序过程中无须比较关键字。分配排序可分为箱排序和基数排序两类。

箱排序（Bin Sort）也称**桶排序**（Bucket Sort），其基本思想是：设置若干个箱子，依次扫描待排序的记录 R[1]、R[2]、…、R[n]，把关键字等于 k 的记录全都装入到第 k 个箱子（分配），然后，按序号依次将各非空的箱子首尾连接起来（收集）。例如，要将一副混洗的 52 张扑克牌按面值 A<2<…<J<Q<K 排序（不分花色），需设置 13 个“箱子”，排序时依次将每张牌按面值放入相应的箱子里，然后依次将这些箱子首尾相接，就得到了按面值递增序排列的一副牌。

显然，在箱排序中，箱子的个数 m 取决于关键字的取值范围。在排序中分配的时间是 O(n)，收集的时间为 O(m+n)（若用链表存储待排记录，则收集的时间为 O(m)），所以箱排序的时间为 O(m+n)。若关键字的取值范围很大，如 $m=O(n^2)$，则箱排序效率很低。

基数排序（Radix Sort）是对箱排序的改进和推广。箱排序只适用于关键字取值范围较

小的情况。否则，所需箱子的个数 m 以及清箱和连接箱子的时间均太多。如果注意到关键字的结构特点，就可使这一问题大为改观。

例如，对前述数据表 49, 38, 49′, 91, 27, 03, 97, 49″，n=8，由于关键字 k_i 由两位数组成，所以，我们可将其分解，先按个位数（由 k_i%10 得到）进行箱排序，然后再按十位数（由 k_i/10 得到）进行箱排序，这样只需标号为 0, 1, …, 9 的 10 个箱子，而不是 0 到 99 的 100 个箱子！第一趟箱排序时，先将顺序扫描到的记录按关键字的个位数装箱，即把 49 装入 9 号箱、38 装入 8 号箱、……、49″装入 9 号箱，其结果如图 7.12（a）行所示；然后依箱号递增顺序将各非空箱首尾相连即得第一趟排序结果：

91，03，27，97，38，49，49′，49″

显然这一序列已按个位有序。

第二趟箱排序前需先清空各箱子，然后，顺序扫描第一趟的结果，并将扫到的记录按关键字的十位数装箱（若无十位数则将十位数看作零）。其结果如图 7.12（b）行所示，把非空箱连接后即得到最终的有序序列：

03，27，38，49，49′，49″，91，97

因为箱子个数 m 的数量级不大于 O(n)，所以，上述排序的时间复杂度是 O(n)。

箱号	0	1	2	3	4	5	6	7	8	9
第一次装箱(a)		91		03				27, 97	38	49, 49′ , 49″
第二次装箱(b)	03		27	38	49, 49′ , 49″					91, 97

图 7.12 两次装箱（分配）

一般地，将数据表中各记录 R[i]的关键字看成一个 d 元组（$k_i^1, k_i^2, \cdots, k_i^d$），每个分量的取值范围相同 $C_1 \leqslant k_i^j \leqslant C_r$（$1 \leqslant j \leqslant d$），可能取值的个数 r 称为基数。基数的选择和关键字的分解视关键字的类型而异，如关键字是十进制整数，则可取基数为 r=10、C_1=0、C_{10}=9，d 为关键字的最大位数。若关键字是由小写英文字母组成的字符串，则可取 r=26、C_1='a'、C_{26}='z'，d 为字符串的最大长度。

基数排序的基本思想是：从低到高依次按关键字的各分量进行箱排序，即首先按 k_i^d 进行箱排序，再按 k_i^{d-1} 进行箱排序，……，最后按 k_i^1 进行箱排序（$1 \leqslant i \leqslant n$）。每趟箱排序所需箱子的个数就是基数 r。显然，这个过程相当于对多关键字进行 LSD 排序。

图 7.12 所示的例子，就是一个基数 r=10、分量数 d=2 的基数排序。

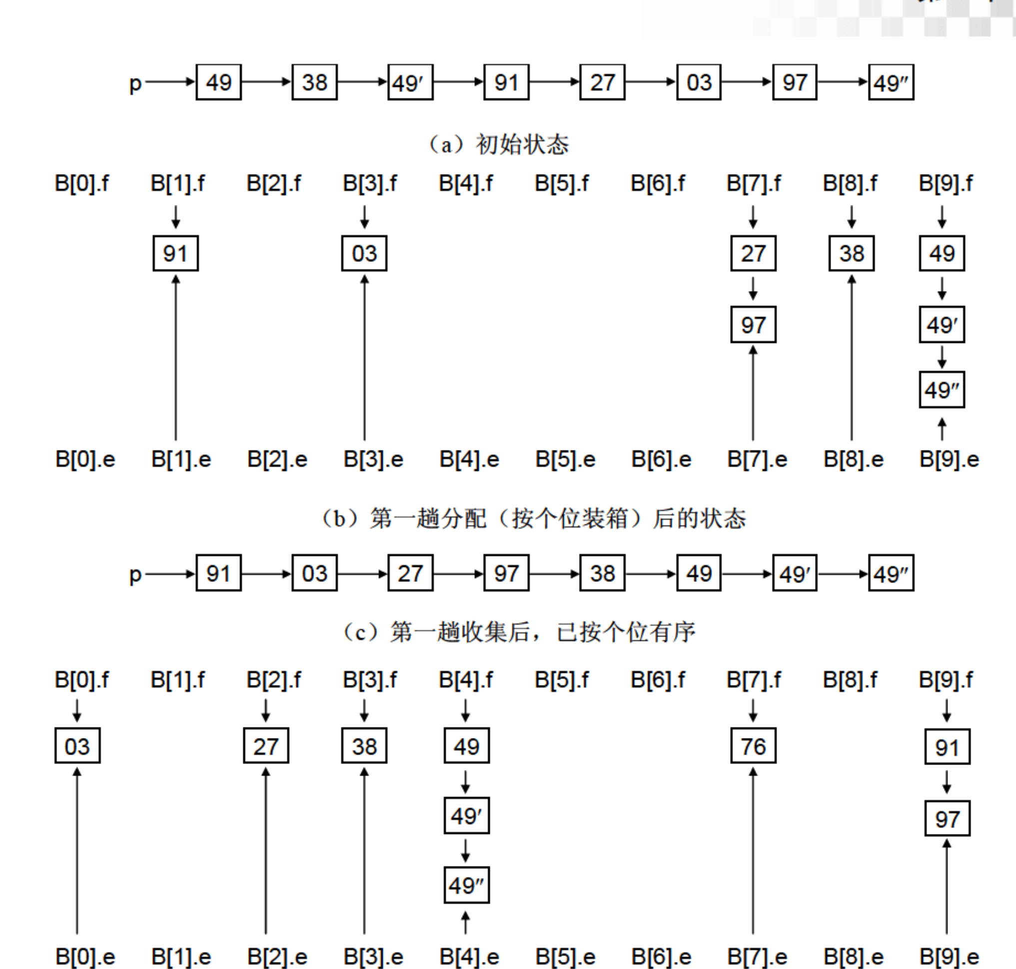
注意，由于每趟排序的箱子是共用的，所以除第一趟外，其他每趟排序前要先清空各箱子。另外，每个箱子内的数据在装箱和收集时的顺序应该相同，否则排序结果会不正确。例如，图 7.12 第二次装箱时，第 9 号箱子的两个数据中 91 先于 97 装箱，在取出时若先取 97 再取 91，结果就错了。所以每个箱子的数据应按先进先出的原则，即按队列存放。

由于每个箱子所装数据的个数是可变的，适合用链表来装箱，所以每个箱子应设计成一个链队列。下面用静态链表作为数据表的存储结构，并且不另设各链队列的结点空间，而利用静态链表中的结点作为链队列中的结点，这样只需修改指针即可完成分配（装箱）和收集（连接箱子）的任务，故又称这种排序为**链式基数排序**。

与基数排序有关的类型定义、变量说明以及算法如下：

```
const int d=3;       //假设分量数为 3
const int r=10;      //假设基数为 10，也即箱子个数
typedef struct {
  int f,e;           //队列的头、尾指针
} queue;             //队列类型
typedef int datatype;
typedef struct {
  datatype key[d];   //关键字由 d 个分量组成
  othertype other;   //记录的其他域
  int next;          //静态链域
} rectype;           //记录结点类型
int RadixSort(rectype R[],int n) {
//对 R[0]~R[n-1]基数排序，返回收集链表的头指针
  int i,j,k,t,p;
  queue B[r];                 //r 个链队列，每个都是一个箱子
  for(i=0;i<n-1;i++)          //将 R[0]~R[n-1]链成一个静态链表
    R[i].next=i+1;
  R[n-1].next=-1;             //将初始链表的终端结点指针置空
  p=0;                        //p 指向链表的第一个结点
  for(j=d-1;j>=0;j--) {       //进行 d 趟箱排序
    for(i=0;i<r;i++)          //清空箱子
      B[i].f=B[i].e=-1;
    while(p!=-1) {            //扫描链表，分箱
      k=R[p].key[j];          //按第 j 个分量分配，k 为箱子号
      if(B[k].f==-1) B[k].f=p;//B[k]为空箱,将 R[p]链到箱头
      else R[B[k].e].next=p;//将 R[p]链到箱尾
      B[k].e=p;               //修改箱子的尾指针
      p=R[p].next;            //扫描下一个记录
    }
    i=0;
    while(B[i].f==-1) i++;  //找第一个非空的箱子
    p=B[i].f;                 //p 为收集链表的头指针
    t=B[i].e;
    while(i<r-1) {
      i++;                    //取下一个箱子
      if(B[i].f!=-1) {        //连接非空箱
        R[t].next=B[i].f;
        t=B[i].e;
      }
    }
    R[t].next=-1;             //本趟收集完毕，将链表的终端结点指针置空
  }
  return p;
}
```

排序结束后，结果仍在数组 R 中，指针 p 指明排序后第一个记录的下标。例如，对前述取值范围为 0 至 99 之间的一组关键字序列：49, 38, 49′, 91, 27, 03, 97, 49″，执行算法 RadixSort，基数 r=10、C_1=0、C_2=9、d=2，其排序过程如图 7.13 所示。



（a）初始状态

（b）第一趟分配（按个位装箱）后的状态

（c）第一趟收集后，已按个位有序

（d）第二趟分配（按十位装箱）后的状态

p → 03 → 27 → 38 → 49 → 49′ → 49″ → 91 → 97

（e）第二趟收集后，已按十位和个位有序

图 7.13　基数排序示例

在上述算法中，没有关键字的比较和记录的移动，只是扫描链表和进行指针赋值，所以排序的时间主要耗费在修改指针上。其中，将 R 初始化成一个静态链表的时间是 O(n)；在每趟箱排序中，清箱时间是 O(r)，分配时将 n 个记录装入箱子，时间是 O(n)，收集的时间是 O(r)。因此，一趟箱排序的时间复杂度是 O(r+n)。因为要进行 d 趟箱排序，所以链式基数排序的时间复杂度是 $O(d\times(r+n))\approx O(d\times n)$，若 d 为常数，则时间复杂度为 O(n) [①]。

但要注意，当 n 较小，d 较大时，基数排序并不一定合适。只有当 n 较大、d 较小时，特别是记录的信息量较大时，基数排序最为有效。

基数排序中，每一个记录中增加了一个 next 域，每个箱子增加了一个头尾指针（B 数组），故辅助存储空间开销为 O(n+r)。

基数排序在分配和收集时不会改变相同键值记录的相对位置，故是稳定的。

① 但若 n 个数据不重复，则所需位数至少为 $d=\lceil\log_r n\rceil$，这时复杂性 $O(n\times d)=O(n\lceil\log_r n\rceil)$，即仍是 $O(n\log_2 n)$。

7.7 内部排序方法的比较和选择

由于排序在计算机中所处的重要地位，以及不同场合对排序的特定要求，人们研究出了众多的排序方法。本章前面介绍的几种排序方法，只是已有算法中为数很少的几个。各种排序方法都有其各自的优缺点，应该说没有哪个是绝对最好的。一般说来，比较简单的排序（如直接插入、直接选择、冒泡排序等，一般称为**简单排序**）每次只对相邻的元素比较，前进步伐慢，时间耗费大，时间复杂度为 $O(n^2)$，但在一些特殊情况下却可取得很好的效果；效率高的算法每次比较产生的作用不仅仅局限于被比较的两个元素，而是多个甚至一半左右的元素，但它们对数据量小的情况并不一定合适。

在实际中如何进行选取呢？一般要综合考虑下列因素：

（1）待排序的记录数目。

（2）记录本身信息量的大小。

（3）关键字的结构及其分布情况。

（4）对排序稳定性的要求。

（5）语言工具的条件。

（6）算法本身的难易程度。

（7）辅助空间的大小等。

依据这些因素，可得出如下几点结论：

（1）若 n 较小（如 $n<50$），可采用一些比较简单的排序方法，如直接插入排序或直接选择排序。在这两者中，当记录本身信息量较大时，宜选用直接选择排序，因为它所需记录移动次数较少；否则可用直接插入排序，它一般比直接选择排序略快。

（2）若 n 较大，且键值分布没有规律，应采用一些时间开销比较小的排序方法，如快速排序、堆排序或归并排序，它们的时间复杂度一般为 $O(n\log_2 n)$。快速排序被认为是目前基于比较的内部排序中最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；但堆排序所需的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏情况。这两种排序都是不稳定的，若要求排序稳定，则可选用归并排序，但它所需的辅助空间最多，不过在 n 较大时，所需时间比堆排序少。

实际上，在基于比较的排序方法中，每次比较两个关键字的大小后，仅出现两种可能的转移，因此可用一棵二叉树来描述比较判定的过程，即排序问题的**判定树**。n 个关键字有 n!种排列，每种排列最后都要作为一个叶子（最终结果）出现在判定树中，即判定树至少有 n!个叶子（有些叶子可能为空或对应的排列相同，取决于具体的算法）。在高度为 h 的二叉树中，叶子结点数不超过 2^{h-1}，反之，有 n!个叶子的二叉树高度至少为 $\lceil\log_2(n!)\rceil+1$。这就是说，最多比较次数（即最坏情况下的比较次数）至少为 $\lceil\log_2(n!)\rceil=O(n\log_2 n)$，这就是最坏情况下的最好时间复杂度。由此还可证明，当 n 个关键字随机分布时，平均比较次数也至少为 $O(\log_2(n!))=O(n\log_2 n)$，这也就是说，时间复杂度为 $O(n\log_2 n)$ 的排序方法基本上是基于比较的排序算法中效率最高的了。

（3）若 n 很大，且关键字有明显结构特征，如字符串和整数，这时可考虑箱排序和基

数排序。由于箱排序和基数排序只需一步就会引起 m 种可能的转移，即把一个记录装入 m 个箱子之一，它们一般可能在 O(n)时间内完成排序。特别地，当记录的关键字位数较少且可以分解时，采用基数排序较好。但如果关键字的取值范围属于某个无穷集合（例如实数型关键字），就无法使用箱排序和基数排序，只能借助于“比较”的方法来排序。

（4）若数据表的初始状态已按关键字基本有序，则可选用直接插入排序或冒泡排序，这时的时间复杂度可能达到 O(n)量级。

（5）若对稳定性有要求，则可用一些稳定的排序方法，如直接插入排序、冒泡排序、归并排序和基数排序等。但是，若是对主关键字排序，则排序方法是否稳定无关紧要。

（6）有的语言（如 FORTRAN、BASIC 等）没有提供指针和递归，则一般只能选择不需要这些技术的排序算法。

（7）前面讨论的排序算法，除基数排序外，都是在一维数组上实现的。当记录本身信息量较大时，为避免耗费大量时间移动记录，可以用链表作为存储结构（采用静态链表可节省指针空间）。直接插入、直接选择、冒泡排序、归并排序等都易于在链表上实现，其中（链）表归并排序由于比较次数最少，更有优势。

但快速排序和堆排序等在链表上难于实现，这时可以提取关键字建立索引表，然后对索引表进行排序。但更简单的方法是进行“**地址排序**”：引入一个数组 t[n]作辅助表，存放各个记录的指针（下标）。排序前，令 t[i]=i（1≤i≤n）；当算法中要求交换 R[i]和 R[j]时，只需交换相应指针 t[i]和 t[j]即可（即由交换记录转换为交换地址）。排序结束后，所有指针将按相应记录键值递增的顺序排列，即向量 t[n]指示了记录之间的顺序关系：

R[t[1]].key≤R[t[2]].key≤…≤R[t[n]].key

如果要求最终记录在物理上递增排列：

R[1].key≤R[2].key≤…≤R[n].key

则只要按索引表或辅助表所规定的次序重排各记录即可，重排的时间复杂度是 O(n)。

最后，为使大家对各种排序方法有一个具体的认识，以及比较一下相同复杂度算法间的相对性能，这里引用文献[4]的一组实验数据，见表 7.1。计算条件是奔腾III处理器和 Windows 98 操作系统，待排序值是 32 位（bit）的随机整数，分别对不同规模的序列对比计算。由于记录不大，移动次数少的算法如选择排序没有显示出其优势。有些排序有两组，其中希尔排序分别对应增量每次折半和每次除以 3；快速排序的第一组是一般的递归算法，第二组是非递归的且不划分长度小于 10 的子序列，最后调用插入排序；归并排序的第一组是一般的递归算法，第二组采用了监视哨技术，并对长度小于 10 的子序列采用选择排序进行处理；基数排序不按十进制处理，而是每次处理 4 个或 8 个二进制位（前者相当于按十六进制处理），但取出有关位时没有采用移位运算。

从表 7.1 可见，时间复杂度为 O(n^2)的算法确实不适合表长较大的情况，其中插入排序的效果略好一点；当规模较大时，希尔排序明显优于 O(n^2)类算法；快速排序，特别是改进后的快速排序是所有算法中最出色的；同类算法的改进并不能有数量级上的变化，但相对速度可能有 50%的提高。基数排序的效果并不突出，但较大的基数可减少分配的趟数，对总的时间可能是有益的。

表 7.1　排序算法实验比较

方法 \ n	10	100	1K	10K	100K	1M	10K 正序	10K 逆序
插入排序	.0011	.033	2.86	352.1	47241	–	0.0	803.0
冒泡排序	.0011	.093	9.18	1066.1	123808	–	513.5	812.9
选择排序	.0011	.072	5.82	563.5	69437	–	577.8	560.8
希尔排序	.0011	.033	5.50	9.9	170	3080	2.8	6.1
希尔排序 / O	.0011	.028	5.50	9.4	160	2800	1.6	4.4
快速排序	.0017	.022	0.33	3.8	49	600	1.7	2.2
快速排序 / O	.0005	.016	0.27	3.3	44	550	1.7	1.6
归并排序	.0027	.049	0.61	8.2	105	1430	6.0	6.1
归并排序 / O	.0005	.022	0.33	4.4	65	930	2.7	3.8
堆排序	.0016	.028	0.38	60.0	94	1650	5.0	5.0
基数排序 / 4	.0500	.467	4.66	47.2	484	4950	47.2	47.2
基数排序 / 8	0429	.252	2.31	23.6	241	2470	23.6	23.6

7.8 外部排序简介

以上各节讨论的排序统称为内部排序，整个排序过程中不涉及数据的内、外存交换，待排序的记录全部存放在内存中。但在许多实际问题中，数据表的记录很多、信息量庞大，无法将整个数据表的所有记录同时调入内存进行排序，只能将数据表存放在外存上，我们称这种排序为外部排序。外存上的数据组织一般称为**文件**，外存文件主要有磁盘文件和磁带文件两大类，它们排序的基本步骤类似，主要不同之处在于初始归并段在外存储介质中的分布方式，磁盘是直接存取设备，磁带是顺序存取设备。

外部排序的实现，除了依靠数据的内外存交换外，基本方法是“内部归并”。首先，根据内存的大小，将待排序的记录 R_1、R_2、…、R_n 分成若干段。依次读入每段的记录，利用前述内部排序方法进行内部排序。这些经过排序的有序段通常称为**顺串**（Run）或归并段，再将其写入外存。这样，在外存上就得到了若干个初始顺串。最后，对这些顺串进行归并，使顺串的长度逐渐增大，直至全体待排序的记录成为一个顺串为止。由此可见，外部排序由两个相对独立的阶段组成：生成初始顺串（初始归并段）以及对顺串（归并段）进行归并。

归并的思想并不复杂，最简单的归并方法是类似于前述 Merge 算法的二路归并。假设内部排序产生的初始顺串有 m 个，进行两两归并后就得到 $\lceil m/2 \rceil$ 个较大的顺串，这就是外排序的第一趟归并。n 个记录 m 个初始顺串需经过 $\lceil \log_2 m \rceil$ 趟归并才能完成外部排序，每一趟需进行全部 n 个记录的内外存交换。

外部排序的时间由三部分组成：内部排序的时间；外存信息读写的时间；内部归并的时间。由于外存信息读写的时间比记录的内部排序和归并所需的时间大得多，因此提高外部排序效率的关键在于减少数据内外存交换的次数，即减少归并的趟数。显然，采用多路归并可减少归并趟数：对 k 路归并，归并趟数为 $\lceil \log_k m \rceil$。另外，减少初始顺串数 m 也可减少归并趟数。在具体实现时有些问题需要解决，以下简单做些介绍。

7.8.1 磁盘排序

磁盘是直接存取设备，读写一个数据块的时间与当前读写头所处的位置关系不大。

1．败者树

k 路归并时，需不断从 k 个归并段的当前记录中选出最小者，若采用直接选择排序的思想，则每次都需 k−1 次比较（已归并完的段当前记录置为∞）。对全部 n 个记录要选择 n 次[①]，于是一趟归并完要比较 n(k−1)次，则$\lceil \log_k m \rceil$趟归并的总比较次数为$\lceil \log_k m \rceil n(k-1)$，将它分成第一次选择和以后 n−1 次选择（以便与下面的树形选择对比），则结果为：

$$\lceil \log_k m \rceil (k-1) + \lceil \log_k m \rceil (n-1)(k-1) = \lceil \log_2 m \rceil \frac{k-1}{\lceil \log_2 k \rceil} + \lceil \log_2 m \rceil (n-1) \frac{k-1}{\lceil \log_2 k \rceil}$$

由于$(k-1)/\lceil \log_2 k \rceil$随 k 增加而增加，特别是第二项的系数$\lceil \log_2 m \rceil (n-1)$较大，所以增加归并路数 k 会导致内部排序时间增大，甚至抵消减少访外次数而赢得的时间。

实际上，每次的 k−1 次比较中，很多在前一次找最小时就已经比较过了，由于没有将结果保留下来，导致许多重复比较。若采用树形选择排序的思想，则每趟归并时，除了建树（即第一次找最小）为 k−1 次比较外，以后每次选择最小时的关键字比较次数最多为$\lceil \log_2 k \rceil$，则$\lceil \log_k m \rceil$趟归并的总比较次数最多为：

$$\lceil \log_k m \rceil (k-1) + \lceil \log_k m \rceil (n-1) \lceil \log_2 k \rceil = \lceil \log_2 m \rceil \frac{k-1}{\lceil \log_2 k \rceil} + \lceil \log_2 m \rceil (n-1)$$

可见，虽然第一项会随 k 增加而增加，但其系数不大，而占主导地位的第二项却与路数 k 无关，从而增加归并路数 k 减少访外次数时不致引起内部排序时间的显著增大，有利于提高外排序的整体效率。

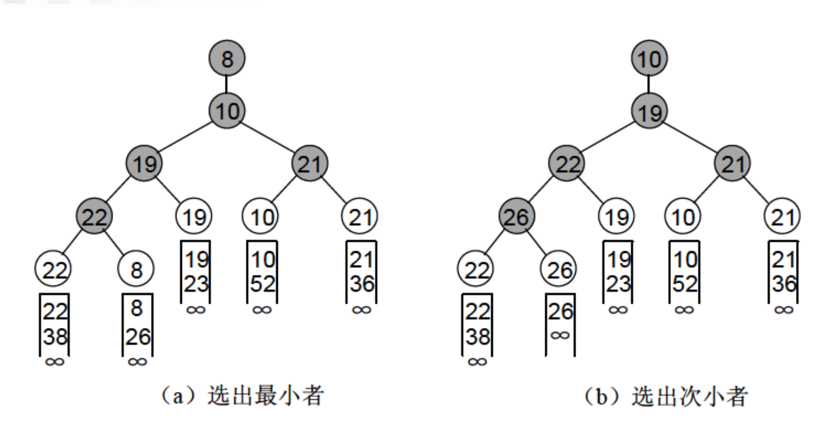
但需指出，k 值并非越大越好。因为系统一般不对外存进行直接的存取，而是通过缓冲区进行的（见下一章），在一定的缓冲区容量下，路数 k 增加则每路的缓冲区就变小，于是限制了内外存交换的数据块的大小，相应地每趟归并时就要更多次地读写数据块，也就增加了访外的次数和时间。所以，k 值过大时，尽管所作的扫描趟数减少，但输入/输出时间仍可能增加。k 的最优值与可用缓冲区大小及磁盘的特性参数等有关。

这里的选择树一般采用**败者树**（Tree of Loser）：双亲结点存放败者（的指针），而胜者去参加高一层的比较，并在原根结点之上附加一个结点存放最优者（的指针）。若双亲存放的是胜者则称**胜者树**（如图 7.7）。与胜者树相比，败者树输出最优者后，对树的修改容易些：新记录从原最优者所在的叶子开始，不断与双亲比较，败者存放到双亲结点，胜者继续与上一级双亲比较，此过程一直进行到根，最后将新的最优者存到附加结点上。可见，该过程只找双亲结点而不必找兄弟结点（也就不必区分左兄弟还是右兄弟了）。

图 7.14（a）所示为从 5 个归并段的当前记录“22, 8, 19, 10, 21”中找最小的败者树，其中∞表示段结束（可防止归并过程中某个归并段变空，即保持归并段数不变，当最后最优者为∞时结束）。图 7.14（b）所示为输出最小记录后，将该段下一个记录 26 替换掉原来最小的叶子 8，然后从该叶子到根，自下而上比较和调整，选出次小关键字 10。

---

① 本来对 n 个数据选择排序，只需选择 n−1 次，但这里的数据分布在 k 个归并段上，即使剩最后一个，也并不知道在哪个段中，仍要选择一次，即需选择 n 次。



（a）选出最小者　　（b）选出次小者

图 7.14　利用败者树选最小记录

注意，双亲存放的并不是两个孩子中的败者（较大者），而是两个孩子所在子树的胜者中的败者（较大者），如图 7.14(a)中阴影结点 10 存放的不是其孩子 19 和 21 中的较大者 21，而是孩子所在子树的胜者 8 和 10 中的败者 10。

败者树初始化时，可先令所有的非终端结点为空（指针），然后从各个叶子出发向上调整非终端结点：若双亲为空，则存放胜者，本叶子调整结束；否则存放败者，胜者继续向上调整。这里判断双亲为空时是指针比较，判断胜负时才是关键字比较，共 k−1 次[①]。

2．置换-选择排序

为了减少初始归并段数 m，就需要使每段的长度尽量长。初始归并段的获得，最简单直接的方法是根据内存和物理页块的大小每次读入若干记录，内排序后输出得到一个初始归并段；这个过程进行下去，最终所有记录便被划分整理成了若干初始归并段。但这样得到的各初始段的大小是相同的（最后一段可能略小），且受内存和物理页块大小的限制，长度难以增加。

一个简单的改进就是每输出一个最小记录后，马上再输入一个记录，这样边输出边输入，就可使初始归并段足够长。但是有个问题，就是若新输入的记录小于已输出的记录，由于不能对文件中的记录进行插入和移动，则该记录应划入下一个归并段，这可将其标记为新的归并段，待前一归并段的记录全部输出后，再用同样的方法形成新的归并段。

其中，在内存记录中找最小者可采用前述败者树，用新输入的记录"置换"刚输出的最小记录后再进行选择。为了正确输出前后两段的记录，将结点比较规则改为：段号不同时段号小者胜，段号相同时关键字小者胜。

以上过程称作**置换-选择排序**（Replacement-selection Sorting），它的特点是在形成初始归并段的过程中，选择（最小或最大关键字）和输入、输出交叉或平行进行。显然，该方法得到的初始归并段长度一般不同，但可证明，当输入文件的关键字随机分布时，初始归并段的平均长度是内存工作区大小的两倍[②]。

---

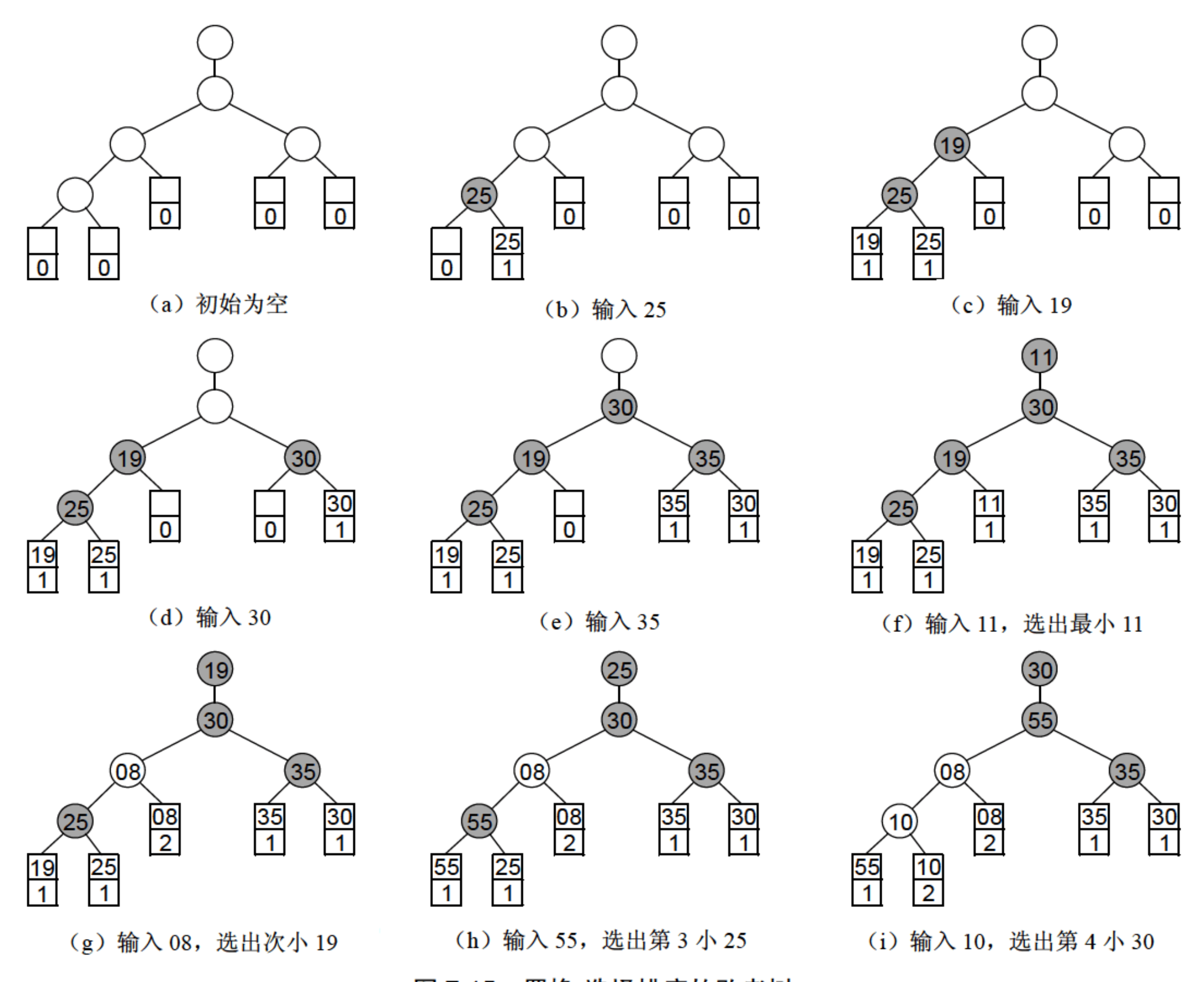
① 初始化时，也可先令所有的非终端结点指向某个辅助结点，其中数据≤所有叶子可能的关键字，然后从各个叶子出发向上调整。这时要涉及和辅助结点的内容比较，关键字比较次数多于 k−1 次。

② 设工作区有 w 个关键字，至少它们都将归并到同一段（即归并段长度最小为 w）。对其中任一位置，关键字被选中输出后，新输入的关键字比它小或大的概率相同，都为 1/2；若是后者则它以后仍将归并到当前段，即新关键字属当前段的概率为 1/2；类似，该关键字被选中输出后再输入的关键字仍属于同一段的概率则为$(1/2)^2=1/4$，依次类推。于是每个位置对当前段贡献的关键字个数为 $1+1/2+1/4+\cdots=2$，故段长平均为 2w。

由于增加了段号比较规则，置换-选择排序的败者树初始化时，不必将所有叶子都读入后再建立，而是边读入边建立：令初始败者树所有记录的段号为 0（实际没有记录），然后逐个读入叶子，自下而上调整败者树，由于段号为 1，相对于原段号 0 的记录自然为败者，从而逐个填充到败者树的各结点中。

图 7.15 表示依次输入 25, 19, 30, 35, 11, 08, 55, 10, …时，置换-选择排序中败者树的部分变化过程，其中用阴影框和空白框区分不同段的结点。

顺便指出，置换-选择排序也可用堆实现：将堆顶输出，在堆顶填入新记录，将结果调整为堆；但若新记录小于原堆顶，则与堆底交换后再调整（新堆底为下一组，不参与本轮调整）。然后再输出、输入、调整。当然，前述 k 路归并也可用堆实现。



(a) 初始为空　(b) 输入 25　(c) 输入 19

(d) 输入 30　(e) 输入 35　(f) 输入 11，选出最小 11

(g) 输入 08，选出次小 19　(h) 输入 55，选出第 3 小 25　(i) 输入 10，选出第 4 小 30

图 7.15　置换-选择排序的败者树

3．最佳归并树

由置换-选择排序得到的初始归并段长度一般不同，在 k 路归并时，不同归并段的组合将导致不同的访外次数。归并过程可用 k 叉树来描述：每 k 个段归并后得到一个新的段，将新段作双亲，原 k 段作其孩子，最后成为 1 段，即树根。这个 k 叉树称为归并问题的归并树（判定树）。图 7.16 表示的是对长度分别为 0, 0, 12, 9, 5, 14, 17, 13, 10, 11, 15, 8, 16 的初始归并段进行 4 路归并的两种方案。其中长度为 0 的是空归并段。

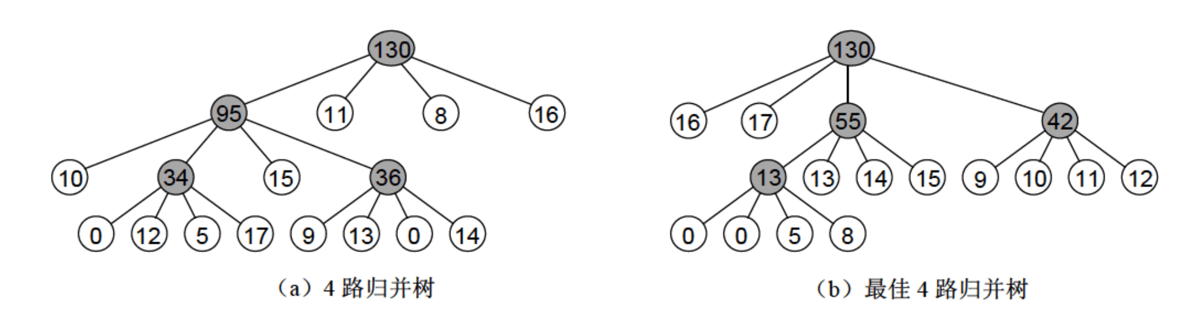


图 7.16　4 路归并树示例

在归并树中，叶子为初始归并段，结点的权为段长（记录数），叶子到根的路径长度表示归并趟数，非叶子结点代表归并出来的新归并段，则归并树的带权路径长度 WPL 即为归并过程中的总读记录数，于是归并过程中总的记录读写次数为 2×WPL。图 7.16(a)的 WPL 为：

$$WPL=(0+12+5+17+9+13+0+14)\times3+(10+15)\times2+(11+8+16)\times1=295$$

不同的归并方案所对应的归并树的带权路径长度各不相同。为了使得总的读写次数达到最少，需要改变归并方案，重新组织归并树。为此，可将哈夫曼树的思想扩充到 k 叉树上：在归并时，让记录数少的归并段先归并，记录数多的归并段后归并，就可以建立总的读写次数最少的最佳归并树。图 7.16(b)就是一棵最佳归并树，它的 WPL 为：

$$WPL=(0+0+5+8)\times3+(13+14+15+9+10+11+12)\times2+(17+16)\times1=240$$

为了分析和处理方便，一般使归并树为严格三叉树（或称正则三叉树，除叶子外，其他结点的度都为 3），这可能需要补充若干空归并段。设 0 度结点有 n_0 个，k 度结点有 n_k 个，易知 $n_0=(k-1)n_k+1$，于是 $n_k=(n_0-1)/(k-1)$。如果该式能整除，则不需补充空归并段；否则设 $(n_0-1)\%(k-1)=m\neq0$，则将 n_0 增加(k−1)−m 即可整除，即补充(k−1)−m 个空归并段即可。在图 7.16 所示例子中，非空段 $n_0=11$，k=4，m=(11−1)%(4−1)=1，故需补充(k−1)−m=2 个空归并段。

以上记录的读写是指逻辑记录，若每个物理块存放 1 个记录，则逻辑读写和物理读写是一致的。若每个物理块存放多个记录，则可按各段所占的物理块数进行类似的分析。

7.8.2　磁带排序

磁带排序的过程也是先生成初始归并段，再反复归并直到全部归并成一段。但磁带是顺序存取设备，不能在同一磁带上进行多路归并（否则寻找各归并段的读写位置会引起大量的进带、倒带，效率极低），即磁带归并及排序需要多台磁带机。另外，各归并段在不同磁带以及同一磁带的不同分布情况对排序效率影响极大。

磁带的 k 路归并排序至少要 k+1 台磁带机：

（1）形成初始归并段时，1 台作输入，k 台作输出。根据内存和页块的大小每次读入若干记录，内排序后将结果轮流写到其他 k 条带上。显然，也可用置换-选择排序生成长度不等的初始归并段，但在磁带上不能进行最佳归并排序。

（2）进行 k 路归并时，k 台作输入，1 台作输出。这时，归并结果在一条带上，归并

完后要对输出带扫描一遍，将其中的各归并段重新分配到 k 条带上，以便下次归并。

为了避免输出带的这种分配扫描，可采用 2k 台磁带机，其中 k 台作输入，k 台作输出；一趟归并完后，输入带和输出带的作用互换（同时输入带和输出带要反转到开始位置），即各磁带机轮流地作输入和输出。

注意到这样一个事实：如果各磁带上的归并段数不同（称为**非平衡归并**，反之，若各磁带上的归并段数基本相同，则称为**平衡归并**），则在归并过程中，归并段数最少的带在某一时刻就变成了空带，于是该带可用作输出带（退到开始位置），而原输出带已生成的归并段（退到开始位置）连同原剩余归并段一起可用作输入，重新开始归并。这样，某趟归并中不等全部记录归并完，就在已有基础上开始了新的一趟归并。这个过程称作**多步归并**（Polyphase Merging Sorting）。这时，k 路归并可只用 k+1 台磁带机，其中每台轮流成为输出带，输入、输出的转折点是某条输入带变空。

显然，为了使多步归并的趟数最小，必须合理分配各磁带上的初始归并段数。注意到最后一步时，一条带为空，其他各带都仅剩一段，据此往回推算，可发现各磁带上的归并段数与 k 阶 Fibonacci 数列有关。假设初始归并段总数满足（否则补充若干长度为 0 的空段）：

$T=kf_i+(k-1)f_{i-1}+\cdots+2f_{i-(k-2)}+f_{i-(k-1)}$

则各带的初始归并段数应为：

$T_1=f_i+f_{i-1}+\cdots+f_{i-(k-2)}+f_{i-(k-1)}$

$T_2=f_i+f_{i-1}+\cdots+f_{i-(k-2)}$

…

$T_{k-1}=f_i+f_{i-1}$

$T_k=f_i$

k 阶 Fibonacci 数列定义如下：

$$f_n=\begin{cases}0 & n=0,1,2,\cdots,k-2\\ 1 & n=k-1\\ f_{n-1}+f_{n-2}+\cdots+f_{n-k} & n\geqslant k\end{cases}$$

以 k=3 为例，该数列为 0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, …。若初始归并段有 193 个，并采用 3 路归并，则由于 $3f_9+2f_8+f_7=193$，于是各带的初始归并段数为 $T_1=f_9+f_8+f_7=81$，$T_2=f_9+f_8=68$，$T_3=f_9=44$。

习　题　七

7.1　对于给定的一组关键字：

503，087，512，061，908，170，897，275，653，462

分别写出直接插入排序、希尔排序（增量为 5, 2, 1）、冒泡排序、直接选择排序、堆排序、归并排序和基数排序的各趟运行结果。

7.2　相对于树形选择排序，直接选择排序和堆排序有何优缺点？

7.3　本章介绍的排序方法中哪些是稳定的？哪些是不稳定的？简单分析一下不稳定的原因。

7.4 试比较直接插入排序、直接选择排序、快速排序、堆排序、归并排序的时空性能。

7.5 分别给出叶结点数 n 为 4、5、6、7 的树形选择排序示意图。

7.6 在 1 000 个数据中找两个最小的，采用什么方法可使关键字比较次数最少？最少为多少？

7.7 已知 A[1..1000]中存放的是小根堆，现要找其中最大的，至少要几次关键字比较？

7.8 若不要求完全排序，仅需知道某序列中前 s 个最大或最小者，试给出解决方法。

7.9 试给出 n=7 的关键字组：

(1) 使快速排序的第一趟，每个关键字都移动一次。该趟移动次数是多少？n=8 呢？

(2) 使快速排序总的移动次数最少，最少是多少？

(3) 使快速排序总的比较次数最少，最少是多少？

(4) 使快速排序总的比较次数最多，最多是多少？

(5) n=7 时，任何基于比较的排序，最坏情况下至少比较多少次？

7.10* 试证明快速排序的平均时间复杂度为 $O(n\log_2 n)$。

7.11 试写一个通过二分查找来寻找插入位置的插入排序算法（即二分插入排序）。

7.12 试写出采用监视哨技术的希尔排序的每趟插入排序算法。

7.13 设计一个双向冒泡排序算法，即在排序过程中交替改变扫描方向。

7.14 设计用单链表表示的算法：

(1) 直接插入排序。

(2) 直接选择排序。

(3) 冒泡排序。

7.15 试写出算法实现以下功能：将数组 A[1..n]中的元素分成 3 部分：前面为负数，中间为 0，后面为正数，要求较好的时间性能。

7.16 试写出快速排序的另两种划分算法。

7.17 试写出快速排序的非递归算法。

7.18 试写出自顶向下（分治法）的二路归并排序算法，并分析其时间复杂度。

7.19 已知关键字序列$\{k_1, k_2, k_3, \cdots, k_{n-1}\}$是大根堆：

(1) 试写一算法，将$\{k_1, k_2, k_3, \cdots, k_{n-1}, k_n\}$调整为大根堆（提示：从新增点 k_n 调整到根）。

(2) 利用(1)的算法写一个建立大根堆的算法。

(3) 该算法建堆时，最多比较次数是多少？

7.20 为了记录数据的比较过程和结果，是否就一定要用完全二叉树？

7.21* 设磁盘文件中记录的关键字分别为 16, 20, 43, 25, 18, 10, 13, 17, 9, 5。假设工作区可容纳 4 个记录，试用选择树法产生初始归并段。

7.22* 对 7 个长度为 L 的归并段进行 2 路磁带平衡归并排序，试写出归并过程。

7.23* 已知两条磁带上的初始归并段数分别为 15 和 35，如果用 2 路多步归并结果如何？

第8章 查找表

查找又称检索，也是数据处理中经常使用的一种重要的运算，几乎在任何一个计算机系统软件和应用软件中都会涉及。一般而言，各种数据结构都会涉及查找操作，如前面介绍的线性表、特殊线性表（栈、队列、串）、数组、广义表、树与图等均可能需要进行查找操作，但没有作为主要操作考虑，它服从于相应的数据结构。在有些应用中，查找操作上升到主要地位，如使用的频率很高，且所涉及的数据量较大，查找的效率就显得格外重要，尤其在一些实时应答系统中更是如此。所以，为了提高查找效率，要专门设置面向查找/检索的数据结构，即查找表。

本章首先介绍查找的一些基本概念，然后分别对静态查找表、树表和散列表进行介绍。树表中重点介绍二叉排序树，其他还有平衡二叉排序树、B 树、B^+树和空间树表等。

8.1 基本概念

查找的概念我们并不陌生，在日常生活中我们就经常进行各种各样的查找，如查字（词）典以找某个字（词）的含义、查电话号码本以找某个人或某单位的电话、在地图中找某个城市等。一般而言，查找是在大量信息或数据中获得所需要的信息或数据。在数据结构中，查找指的是在若干数据元素（记录）的集合中求出满足某给定条件的记录。这里的“条件”可能是多种多样的，本章中将此条件仅限于与给定值相匹配，于是有如下定义：

查找（Search）就是在 n 个数据元素中找出关键字等于给定值 K 的结点。若找到，则称查找成功，否则称查找失败。一般情况下，查找成功时，返回所找到的记录的位置（指针）或给出该记录的有关信息；不成功时返回一个空指针或给出不成功信息。

这种查找也称为精确查找。其他的查找还有范围查找（查找关键字在某个范围内的记录）、组合查找（查找满足多个条件的记录）、模糊查找等。

和排序类似，查找也有**内查找**和**外查找**之分。前者指涉及的查找对象全部在内存；后者涉及的查找对象很多，不能一次性全部存入内存，在查找过程中，需要访问外存。

与排序类似，评价查找算法的好坏，一般也是考察算法的时间和附加空间耗费，以及算法的复杂程度等。由于附加空间一般都不大，故主要考察查找的时间开销。

由于查找运算的主要操作是关键字的比较，所以，通常把查找过程中对关键字需要执行的平均比较次数（也称为平均查找长度）作为衡量一个查找算法效率优劣的标准，**平均查找长度**（Average Search Length，ASL）定义为：

$$ASL = \sum_{i=1}^{n} p_i c_i$$

其中，n 是结点的个数；p_i 是查找第 i 个结点的概率，若不特别声明，均认为每个结点的查找概率相等，即 $p_1 = p_2 = \cdots = p_n = 1/n$；$c_i$ 是找到第 i 个结点时所需要的比较次数。

进一步，设查找成功的概率为 p，查找不成功的概率为 q=1−p，还可定义总的平均查找长度：

$$ASL = p \cdot ASL_{suc} + q \cdot ASL_{un}$$

查找表（Search Table）是一种以集合为逻辑结构、以查找为核心运算，同时包括其他运算的数据结构。由于集合中的数据元素之间是没有“关系”的，因此在查找表的实现时可不受“关系”的约束，而根据问题的具体情况和要求去组织查找表，以便实现高效率的查找。

查找表上的运算一般有建表、查找、读表元、插入和删除等。但有些查找表在建立后基本上不需要进行插入和删除运算，于是我们在实现和应用时可有针对性地进行处理以提高效率。为此将查找表分成两类：静态查找表和动态查找表。

静态查找表刻画并适应于下列场合：查找表一经生成之后，便只对其进行检索（包括查找和读表元）而不进行修改（插入和删除）；或者进行一段时间的检索之后集中地进行修改。也就是说，作为主要工作的检索与作为次要工作的修改被分为两个不交叉的阶段分别进行，每个阶段只做一项工作。相反，**动态查找表**刻画并适应于下列场合：检索与修改交叉进行，无法分成两个不相交的阶段。

静态查找表包括以下 3 种基本运算：

（1）建表 CREAT(ST)。加工型运算，生成一个由用户给定的若干数据元素组成的静态查找表 ST。

（2）查找 SEARCH(ST, K)。引用型运算，在查找表 ST 中查找键值等于 K 的数据元素的位置；若没有，结果为一特殊标志。

（3）读表元 GET(ST, i)。引用型运算，取查找表 ST 中 i 位置上的数据元素的值。

动态查找表包含以下 5 种基本运算：

（1）查找。同静态查找表。

（2）读表元。同静态查找表。

（3）插入 INSERT(ST, K)。加工型运算，在查找表 ST 中插入键值等于 K 的数据元素，若原来已有键值等于 K 的数据元素，则不插入。

（4）删除 DELETE(ST, K)。加工型运算，在查找表 ST 中删除键值等于 K 的数据元素。

（5）初始化 INITIATE(ST)。加工型运算，设置一个空的动态查找表。

注意，动态查找表的基本运算中没有建表，因为它的表结构是在查找过程中动态生成的，即当查找不成功时就插入相应的结点。另外，插入、删除运算也会修改表结构，所以也可按表结构在查找过程中是否会变化来定义（或区分）静态查找表、动态查找表。

在存储实现上，4 种基本的存储结构查找表都可采用，如静态查找表主要采用顺序存储和索引存储；树表主要采用链式存储；散列表采用散列存储。因为查找是对已存入计算机的数据所进行的操作，所以数据的具体组织形式，即存储结构对查找速度会有决定性的影响。

由于两种查找表的实现涉及的问题差别很大，下面各节将分别讨论。

 8.2 静态查找表实现

静态查找表通常是将数据元素组织为一个线性表，可以采用顺序存储，也可以采用链式存储。静态查找表的建表和读表元运算比较简单，下面主要研究查找运算在不同存储表示下的实现。本节主要介绍 3 种查找方法：顺序查找、二分查找和分块查找。注意，它们同样也可用于一般线性表上进行查找。

8.2.1　顺序表上的查找

静态查找表最简单的实现方法是采用顺序存储结构，并在此基础上实现其基本运算（这里只考虑查找的实现），我们把这种实现的查找表称为**顺序（查找）表**，它和我们在线性表中介绍的顺序表是有区别的：后者的结点之间有逻辑关系，结点的存储位置在次序上与逻辑次序一致；而这里结点之间没有逻辑关系，结点的存储位置不表示逻辑关系，它们可以是任意的。由于两者的查找方法相同，我们这里也不严格区分。顺序表的类型定义如下：

```
const int maxsize=100;      //顺序表的容量(表长)，假设为 100
typedef struct {
   keytype key;              //关键字项
   othertype other;          //其他域（根据实际情况设置或取消）
} record;                    //查找表的结点类型
typedef struct {
   record data[maxsize+1];   //从 data[1]开始存放数据
   int n;                    //顺序表实际元素个数
} sqtable;                   //顺序表的类型
sqtable R;                   //顺序表
```

这里，从数组 data 的 1 号单元开始存放数据，0 号单元未用，但可用于设置“监视哨”，见下面的叙述。

在顺序表上一种最简单直观的查找方法是**顺序查找**（Sequential Search）。它的基本思想是：从表的一端开始，顺序扫描查找表，依次将扫描到的结点关键字和给定值 K 相比较，若它们相等，则查找成功，并给出数据元素在表中的位置；若全部元素扫描完后，仍未找到关键字等于 K 的结点，则查找失败。

具体查找时，可从前向后进行，也可从后向前进行，第 2 章顺序表的定位（按值查找）算法采用的就是从前向后的顺序查找。这里再考察一下从后向前的顺序查找，算法如下：

```
int search(sqtable *R,keytype K) {
   int i;
   i=R->n;
   while(i>0 && R->data[i].key!=K) i--;
   return i;                 //查找成功时返回 K 在表中的序号，否则返回 0
}
```

其中 while 循环结束时有两种可能：查找成功或失败，本应分不同情况进行返回，即查找成功时返回找到的位置 i（i>0），失败时返回 0。但查找失败时 i 正好也为 0，所以不论查找成功与否，都可用“return i;”表示。

循环中要检查下标是否越界 i>0，如果将 R->data[0]用来设置监视哨，则可使省去这一条件的检查，从而节省比较的时间。这种技巧在第 7 章直接插入排序中就已用过。带监视哨的顺序查找算法如下：

```
int search(sqtable *R,keytype K) {
   int i;
   R->data[0].key=K;          //设置监视哨
   i=R->n;
   while(R->data[i].key!=K) i--;
   return i;                  //查找成功时返回 K 在表中的序号，否则返回 0
}
```

显然，若找到的记录是 R->data[n]，则比较次数 $c_n=1$；若找到的是 R->data[1]，则比较次数 $c_1=n$；一般情况下，找到顺序表中第 i 个记录时所需的比较次数为 $c_i=n-i+1$。假定每个记录被查找的概率相等，则 $p_i=1/n$，从而

$$ASL=\sum_{i=1}^{n}p_ic_i=\sum_{i=1}^{n}\frac{1}{n}(n-i+1)=\frac{n+1}{2}=O(n)$$

这就是说，查找成功时的平均比较次数约为表长的一半。显然，对于不成功的查找，比较过程一直进行到监视哨结束，比较次数为 n+1（无监视哨时为 n 次，但需下标检查）。

不论成功与否，总的平均比较次数为

$$ASL=p\cdot ASL_{suc}+q\cdot ASL_{un}=p\cdot(n+1)/2+(1-p)\cdot(n+1)=(n+1)\cdot(1-p/2)$$

可得$(n+1)/2\leqslant ASL\leqslant(n+1)$，即不论查找成功与否，平均比较次数最少$(n+1)/2$次，最多 n+1 次。

不难发现，顺序查找也可用于链式组织的查找表（对单链表从前向后进行），而链表适合于插入和删除运算，所以顺序查找法也可用于动态查找表。

在很多情况下，查找表中数据元素的查找概率是不相等的。以从前向后进行的顺序查找为例，不等概率的平均查找长度为：

$$ASL'=\sum_{i=1}^{n}p_ic_i=p_1+2p_2+\cdots+(n-1)p_{n-1}+np_n$$

显然，$p_1\geqslant p_2\geqslant\cdots\geqslant p_n$ 时 ASL′ 达到最小值。这需要将各结点按查找概率由大到小的次序存放。但各结点的查找概率事先一般并不知道，为了提高查找效率，可对算法做些修改。

方法一：每个结点增加一个查找频率计数器，每查找一次，计算器增 1，如果某结点的查找频率超过了其前面（指查找开始方向的那一侧）的结点，就交换两者的位置。这样查找概率大的结点在查找过程中不断往前移，可减少以后重复查找时的比较次数，但该方法增加了结点的存储空间。

方法二：每当查找成功，就将找到的结点向前移动一步，即和它前面的结点（若存在）交换。这也可使查找概率大的结点不断往前移。

方法三：如果查找表是用链表组织的，还可以将每次找到的结点移到链表的头部（顺序组织时这种移动不方便，要引起大量结点的后移），更利于以后重复查找，这有点类似缓存技术。

顺序查找的优点是算法简单，且对表的结构无任何要求，无论是用向量还是用链表来存放结点，也无论结点之间是否按关键字有序，它都适用。顺序查找的缺点是查找效率低，因此，当 n 较大时，不宜采用顺序查找。

8.2.2　有序表上的查找

一般情况下，关键字之间可能构成某种次序关系，如键值为数值型时，数值的大小就是一种次序关系；若键值为字符型，字典序就是一种次序关系。注意，这里的次序关系不是逻辑关系。若顺序表中各结点按键值的某种次序排列成有序表，则查找时可采用效率较高的一些算法，如二分查找等。在以下讨论中，假设有序表是递增有序的。

1. 二分查找

二分查找（Binary Search）又称**折半查找**，其基本思想是：首先将待查的 K 值和有序表中间位置 mid 上的结点的关键字 key 进行比较，若相等，则查找完成；否则，若 K<key，则说明待查找的结点只可能在有序表中 mid 位置左边的子表中，则在左子表中继续进行二分查找；若 K>key，则说明待查找的结点只可能在 mid 位置右边的子表中，则在右子表中继续进行二分查找。如此进行下去，直到找到关键字为 K 的结点，或者当前的查找区间为空（即查找失败）。这样，每经过一次关键字比较，剩下的查找区间就缩小为原来的一半，二分或折半查找即由此得名。

例如，假设被查找的有序表中关键字序列为：

5，11，23，35，51，64，72，85，88，90，98

当给定的 K 值分别为 72 和 30 时，进行二分查找的过程如图 8.1 所示，其中方括号表示当前的查找区间。

```
序号  1    2    3    4    5    6    7    8    9    10   11
     [5    11   23   35   51   64   72   85   88   90   98]
      ↑                        ↑                        ↑
     low                      mid                      high
      5    11   23   35   51   64  [72   85   88   90   98]
                                    ↑         ↑         ↑
                                   low       mid       high
      5    11   23   35   51   64  [72   85]  88   90   98
                                   ↑↑    ↑
                               low mid  high
```

（a）查找 K=72 的过程（三次关键字比较后查找成功）

```
序号  1    2    3    4    5    6    7    8    9    10   11
     [5    11   23   35   51   64   72   85   88   90   98]
      ↑                        ↑                        ↑
     low                      mid                      high
     [5    11   23   35   51]  64   72   85   88   90   98
      ↑         ↑         ↑
     low       mid       high
      5    11   23  [35   51]  64   72   85   88   90   98
                     ↑↑   ↑
                 low mid  high
      5    11   23] [35   51   64   72   85   88   90   98
                ↑    ↑
              high  low
```

（b）查找 K=30 的过程（三次关键字比较后查找失败）

图 8.1　二分查找过程示例

显然，若表中有多个关键字相同，则找到的只是它们中的某一个，不一定是最前（或最后）的一个（顺序查找则是）。二分查找算法如下：

```
int BiSearch(sqtable *R,keytype K) {//升序折半查找，非递归算法
  int low,high,mid;                //low、high表示当前查找区间的下界和上界
  low=1;high=R->n;
  while(low<=high) {
    mid=(low+high)/2;                    //求中间位置
    if(K==R->data[mid].key) return mid;  //找到
    else if(K<R->data[mid].key) high=mid-1;//下次在前半部分查找
    else low=mid+1;                      //下次在后半部分查找
  }
  return 0;                        //查找失败
}
```

二分查找过程也可以递归地进行，故也可很容易地写出相应的递归算法（略）。

在二分查找中，每次都以表的中点为比较对象，将表分为两个子表，对定位到的子表再进行同样的操作。这一过程可用一棵二叉树来描述：将当前查找区间中间位置上的结点作为根，左子表和右子表中的结点分别组成根的左子树和右子树，而左右子树又按同样的规则建立，……。由此得到的二叉树，称为描述二分查找的**判定树**（Decision Tree）。显然，整个判定树的根是第1次比较的中点结点，若某一步时左子表或右子表为空，则对应的子树为空。

例如，上述11个结点的有序表的二分查找过程可用图8.2所示的判定树表示，树中结点内的数字表示该结点在有序表中的位置。从图中可见，若查找的结点正好是表中第6个结点，则只需进行一次比较；若查找的结点是表中第3或第9个结点，则需进行二次比较；若找第1、4、7、10个结点则各需比较三次；若找第2、5、8、11个结点则各需比较四次[①]。

由此可见，二分查找过程恰好是走了一条从判定树的根到被查结点（或某个空子树）的一条路径，经历比较的关键字个数恰为路径中的结点数（也即路径最后一个结点在树中的层数）。例如，图8.2中根右侧的虚线表示图8.1（a）查找K=72的过程，其中将K分别与结点6、9和7比较，共进行了三次比较后查找成功；根左侧的虚线表示图8.1（b）查找K=30的过程，其中将K分别与结点6、3和4比较，共进行了三次比较后查找失败。

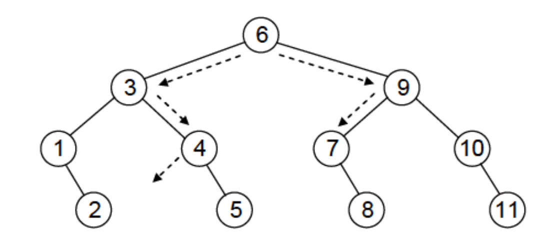


图8.2　11个结点的二分查找判定树及查找过程

注意，空子树实际上对应的是一个范围，如结点4的左子树表示大于R[3]而小于R[4]。

① 这里的“比较”应指对“结点”的比较或**探查**，因为纯粹从关键字比较上看，对每个结点x，查找成功时比较了1次，不成功时又进行小于（或大于）比较，即比较了2次：if(K==x.key)…else if(K<x.key)…else…，参见前述二分查找算法。当然，从汇编语言的角度看，则只需1次关键字比较：cmp K, x.key; je …; jl…。这是因为比较产生的标志位可多次使用（即相同数据不需再次比较，除非标志位被更改）。

易见 n 个结点的判定树中有 n+1 个空子树，分别对应 n 个结点关键字之间的区间范围。

借助判定树，可推出在等概率情况下，二分查找的平均查找长度。由于度小于 2 的结点只可能在最下面的两层上，所以 n 个结点的判定树的高度 h 和 n 个结点的完全二叉树相同，为 $\lceil \log_2(n+1) \rceil$ 或 $\lfloor \log_2 n \rfloor+1$。前 h−1 层为满二叉树，共 $2^{h-1}-1$ 个结点，故第 h 层有 $n-(2^{h-1}-1)$个结点，而第 i 层结点的查找长度为 i，所以平均查找长度为：

$$ASL = \sum_{i=1}^{n} p_i c_i = [1\times 2^0 + 2\times 2^1 + \cdots + (h-1)\times 2^{h-2} + h\times(n-2^{h-1}+1)]/n$$

$$= \frac{n+1}{n}h - \frac{2^h-1}{n}$$

其中级数和 $1\cdot 2^0 + 2\cdot 2^1 + \cdots + k\cdot 2^{k-1} = (k-1)\cdot 2^k + 1$（见附录 E）。注意 $2^{h-1}-1<n\leqslant 2^h-1$，则上式第二项 $1\leqslant \frac{2^h-1}{n} = \frac{2(2^{h-1}-1)+1}{n} < 2+\frac{1}{n} \approx 2$，而第一项 $\frac{n+1}{n}h \approx h$，所以，二分查找的平均查找长度约介于 h−1 到 h−2 之间。而树的高度 h 就是查找时的最大比较次数（不论成功与否），这表明，二分查找的最坏性能和平均性能相当接近。特别地，对满二叉树：

$$ASL = \frac{n+1}{n}h - 1 = \frac{n+1}{n}\log_2(n+1) - 1 \approx \log_2(n+1) - 1$$

对一般情况，也常用该式估计平均查找长度（比准确值略小）。

顺便指出：

（1）判定树本身并没有要求结点编号从 1 开始，如果我们将编号从 0 开始，则只要结点数相同，判定树的结构是完全相同的，仅仅是树中每个结点的序号减 1 而已。

（2）判定树是一个非常重要的概念，也是一个非常重要的辅助分析工具。我们在第 5 章引入了分类过程的判定树；在第 7 章引入了排序过程的判定树；现在又有了查找过程的判定树，而很多其他问题也会有相应的判定树。它们有一个共同特点：结点表示某种比较或判断，分支表示各种可能。

对查找问题，若判定树中每个结点只存放一个关键字（一般用相应元素的编号表示），对该关键字比较后可产生两种或三种情况，即相等、不等，或相等、小于和大于，若使结点本身就表示相等情况，则每次比较后最多产生两个分支，从而判定树为一棵二叉树。这时结点的比较次数就是关键字的比较次数。查找方法不同，判定树也不同，如顺序查找的判定树是单枝树；二分查找的判定树是形态很均衡的二叉树。

一般地，设判定树中有 n 个结点，则其高度最小为 $\lceil \log_2(n+1) \rceil$ 或 $\lfloor \log_2 n \rfloor+1$，与二分查找的判定树相同，这也就是最坏情况下关键字的最少比较次数。对成功的查找，平均查找长度就是所有结点的路径长度和的平均值+1。设想把高度大的结点移动到高度小的空位置，则路径和必定减少。所以最小路径和在树高最小时达到，从而又与二分查找的相同，于是平均查找长度最少为 $O(\log_2 n)$。这些就是基于关键字比较来进行查找的算法所能达到的最好结果，而二分查找就达到了这个最好结果。

显然，若判定树中每个结点存放多个关键字，则判定树为多叉树。这时树的高度和平均查找长度都可能减少到 $O(\log_k n)$。但这是对“结点”比较而言的，而每次对结点比较时一般要进行多次关键字比较（最多比较完结点中的所有关键字，可参见后面将介绍的 B 树），事实上，关键字的比较次数最好仍为 $O(\log_2 n)$。

2．插值查找

折半查找每次以数据集的中点作为比较的对象，并将数据集分割成前后两部分。如果已知两个端点的情况，则我们还可以根据当前待查关键字的大小，利用数学上的两点插值公式确定一个比较准确的分割点，这就是（线性）**插值查找法**（Interpolation Search）。

实际上我们在日常生活中就在不知不觉地使用着这种方法，如查英文词典时，若找一个以 w 开头的单词，我们会在词典的较后部分而不是在开头或中间部分查找，这是因为词典是按首字母顺序排列的，我们要找的 w 肯定位于较后部分。分割点的插值计算式为：

$$\text{mid} = \text{low} + \frac{K - K(\text{low})}{K(\text{high}) - K(\text{low})}(\text{high} - \text{low})$$

其中 low、high、mid 分别为当前查找区间的两个端点和分割点的下标，K(low)、K(high)、K 分别为相应点的键值。插值查找法的实现方法与折半查找很类似：

（1）若 K=K(mid)，则查找成功。

（2）若 K<K(mid)，则置 high=mid−1，继续在左区间查找。

（3）若 K>K(mid)，则置 low=mid+1，继续在右区间查找。

插值查找是平均性能最好的查找方法，但只适合关键字分布比较均匀的情况，其时间性能仍然是 $O(\log_2 n)$①。

3．斐波那契查找

除了采用数据集的中点和线性插值点对数据集进行分割外，还可采用斐波那契（Fibonacci）数列来分割数据集。（二阶）斐波那契数列定义如下：

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n \geqslant 2 \end{cases}$$

即该数列为：0, 1, 1, 2, 3, 5, 8, 13, 21, …。

如果初始待查区间长度 n 正好是某个斐波那契数−1（若不是可在数据集上增设若干虚结点），即 n=F(k)−1，则选取该区间[1, n]内序号为 F(k−1)的点作为分割点，它将数据集分为两个部分：左区间长度为 F(k−1)−1，右区间长度为 n−F(k−1)=[F(k)−1]−F(k−1)=F(k−2)−1。于是，待查值与分割点比较后，不论下一步是继续在左区间查找还是在右区间查找，两个区间的长度仍是某个斐波那契数−1，于是对子区间又可进行同样的处理。当 n 较大时，每次分割出来的两个区间长度之比约为 0.618，即数学上的**黄金分割法**。

一般情况下，设当前查找区间为[low, high]，区间长度为 F(i)−1，则分割点的下标为 mid=low+[F(i−1)−1]。由斐波那契数列的特点可知，对下一步来说，若是左区间，分割点位置将为 F(i−2)；若是右区间，分割点位置则为 F(i−3)。至于每个斐波那契数，除了初始区间的两个外，其他区间可由后向前递推求出。于是，斐波那契查找算法可描述如下。

（1）由区间长度 n 定出 F_k 和 F_{k-1}，其中 $n=F_k-1$；

low=1，high=n；

① 插值查找利用了关键字的分布规律（假设为线性），由函数值估算位置，相当于后文将介绍的散列查找，效率应比只通过关键字比较的算法高些，约为 $O(\log_2\log_2 n)$。但在大 O 表示下，$O(\log_2 n)$也是其一个上界。

（2）若 low>high，查找失败，结束；否则：

　　$mid=low+(F_{k-1}-1)$；

（3）若 K=K(mid)，查找成功，结束。

（4）若 K<K(mid)，则

　　high=mid−1；

　　$F_{k-2}=F_k-F_{k-1}$；

　　$F_k=F_{k-1}$；$F_{k-1}=F_{k-2}$；转（2）；

（5）若 K>K(mid)，则

　　low=mid+1；

　　$F_{k-2}=F_k-F_{k-1}$；$F_{k-3}=F_{k-1}-F_{k-2}$；

　　$F_k=F_{k-2}$；$F_{k-1}=F_{k-3}$；转（2）；

可以证明，斐波那契查找判定树的高度约为 $1.44\log_2 n$[①]，比二分查找的大，故最坏情况下性能比二分查找差，但平均性能仍为 $O(\log_2 n)$。该方法的一个优点是不需乘除法运算。另外，在查找过程中查找位置移动的平均距离比二分查找小（从这点看平均性能比二分查找“好”，但不是比较次数较少）。

上面介绍的 3 种查找方法，效率都较高，但都要求关键字有序并且是顺序存储（从而主要用于静态查找表，因为顺序结构里插入和删除不方便）。

易见，可把它们统一成广义的二分查找：将区间一分为二，分割点分别为区间中点、线性插值点、斐波那契区间点等。

8.2.3　索引顺序表上的查找

索引顺序表是按照索引存储方式构造的一种存储结构，它由两部分组成：一个顺序表和一个索引表。其中顺序表中的元素按块有序；对每一块，在索引表中建立一个索引项，所有索引项顺序存储组成索引表。每个索引项由两部分组成：块中的最大关键字及块的起始位置。由于顺序表是分块有序的，所以索引表是一个递增有序表。

这里所谓“按块有序”或“分块有序”是指：顺序表中的数据可划分为若干子表（块）；每一块中的关键字不一定有序，但前一块中的最大关键字必须小于后一块中的最小关键字。例如，图 8.3 就是满足上述要求的存储结构，其中顺序表 R 有 18 个结点，被分成 3 块，每块中有 6 个结点，第一块中最大关键字 15 小于第二块中最小关键字 17，第二块中最大关键字 35 小于第三块中最小关键字 37。

在索引顺序表上的查找方法是**分块查找**（Blocking Search）或称**索引顺序查找**，它是一种性能介于顺序查找和二分查找之间的查找方法。它的基本思想是：首先，查找索引表，因为索引表是有序表，故可采用二分查找或顺序查找，以确定待查的结点在哪一块；然后在已确定的那一块中进行顺序查找。

例如，在图 8.3 所示的存储结构中，查找关键字等于给定值 K=20 的结点，因为索引表小，不妨用顺序查找方法查找索引表。即首先将 K 依次和索引表中各关键字比较，直到

---

① 该判定树相当于后文将提到的 AVL 树在结点数最少时的情形之一，故高度与后者相同，推导略。

找到第 1 个关键字大于等于 K 的结点，由于 K<35，所以，关键字为 20 的结点，若存在的话，则必定在第二块中；然后，由 ID[2].addr 找到第二块的起始地址 7，从该地址开始进行顺序查找，直到 R[10].key=K 为止。若给定值 K=30，类似地，先确定第二块，然后，在该块中查找，查找不成功，说明表中不存在关键字为 30 的结点。

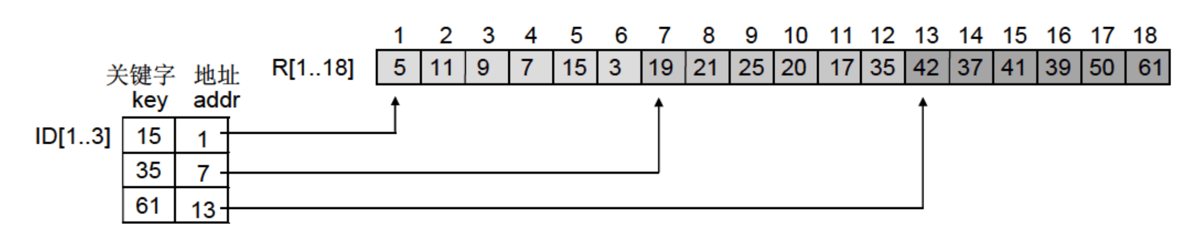


图 8.3 索引顺序表示例

分块查找实际上是两次查找过程，故整个算法的平均查找长度，是两次查找的平均查找长度之和。假设将顺序表分成 b 块，每块的元素个数为 $\alpha_i n$（$1 \leqslant i \leqslant b, 0 < \alpha_i < 1, \sum \alpha_i = 1$）。

首先看 α_i 的选取。在分块数一定时，索引表部分的查找效率不变，而所有块顺序查找效率的平均为 $\alpha_1\left(\frac{\alpha_1 n+1}{2}\right)+\alpha_2\left(\frac{\alpha_2 n+1}{2}\right)+\cdots+\alpha_b\left(\frac{\alpha_b n+1}{2}\right)$，易知该式在各 α_i 相等时最小，即每块均分时效果最好。再看块数 b 的选取。设每块的元素个数为 s，则元素总数 n=b×s（若元素不能完全均分，那么前 b−1 块中结点个数为 $s=\lceil n/b \rceil$，第 b 块的结点数少于 s）。

若对索引表进行二分查找以确定块，则分块查找的平均查找长度为：

$$ASL_{blk}=ASL_{bn}+ASL_{sq}\approx \log_2(b+1)-1+(s+1)/2 \approx \log_2(n/s+1)+s/2$$

近似取 $n/s+1 \approx n/s$，上式在 $s=2/\ln(2) \approx 2.89$ 时有最小值，相应地 $b \approx n/2.89$。但这时块很多，每块元素很少，基本上相当于稠密索引和二分查找，“分块”的意义不大。

若对索引表进行顺序查找以确定块，则分块查找的平均查找长度为：

$$ASL'_{blk}=\frac{b+1}{2}+\frac{s+1}{2}=\frac{1}{2}\left(\frac{n}{s}+s\right)+1 \geqslant \sqrt{n}+1$$

当 $b=s=\sqrt{n}$ 时上式取等号，即 ASL'_{blk} 取极小值 $\sqrt{n}+1$。所以分块查找时如果对索引表进行顺序查找，则将数据均分成 $\sqrt{n}$ 块，每块的结点数为 $\sqrt{n}$ 比较好。

例如，若表中有 10 000 个结点，则应把它分成 100 个块，每块中含 100 个结点。这时分块查找平均做 101 次比较，顺序查找平均做 5 000.5 次比较，二分查找平均做 12.3 次比较（最多 14 次比较）。由此可见，分块查找算法的效率介于顺序查找和二分查找之间。

注意，在实际使用时，每块的大小不一定能做到相同，如一个学校的学生情况很自然地是按系或按班分块，各系或各班的人数就不一定相同。另外，各块的存储区除了内存，还可以是外存（此时的块一般对应外存一个 I/O 读写的物理块）；各块也不一定都要放在同一个向量中，还可将不同块放在不同的向量中。显然，如果将每一块的元素组织成一个链表，分块查找的思想仍然可以使用。

如果要在表中插入或删除记录，只需对该记录所在块进行就可，并且由于块内记录的存放是任意的，插入或删除比较容易，无须移动大量记录。这个特点当不同的块组织成不同的链表，或者组织到不同的向量中时更加明显（各块组织在同一向量中时，可在每块的后部预留一定空间）。可见，分块查找也可用于动态查找表。

分块查找的主要代价是需要增加一个辅助数组的存储空间作索引表，以及将初始表整理（划分）成分块有序的运算。

8.3　树表的查找

上面介绍的 3 种查找法，主要适用于静态查找表，其中以二分查找效率最高，但它要求表中结点按关键字有序。对于动态查找表，在采用链表作存储结构后，虽然也能用上面提到的顺序查找和分块查找，但更好的是采用本节将介绍的几种特殊的树或二叉树作为表的组织方式，在此将它们统称为**树表**。不过下面将看到，建立树表的过程本身也相当于一个排序的过程。树表一般采用链式存储结构以适应结点的经常增加或删除。

8.3.1　二叉排序树

二叉排序树（Binary Sort Tree）又称为**二叉查找树**（Binary Search Tree），它是一种特殊结构的二叉树，其定义为：二叉排序树或者是一棵空树，或者是具有如下性质的二叉树：

（1）若它的左子树非空，则左子树上所有结点的键值均小于根结点的键值。

（2）若它的右子树非空，则右子树上所有结点的键值均大于根结点的键值。

（3）左、右子树本身又各是一棵二叉排序树。

按此定义，二叉排序树中没有键值重复的结点[①]。

二叉排序树有一个重要性质：按中序遍历得到的中序序列是一个递增有序序列。例如，图 8.4 所示的两棵树均是二叉排序树，以其中子图（a）为例，对其进行中序遍历，则得到有序序列：11, 33, 36, 40, 42, 55, 56, 58, 60。

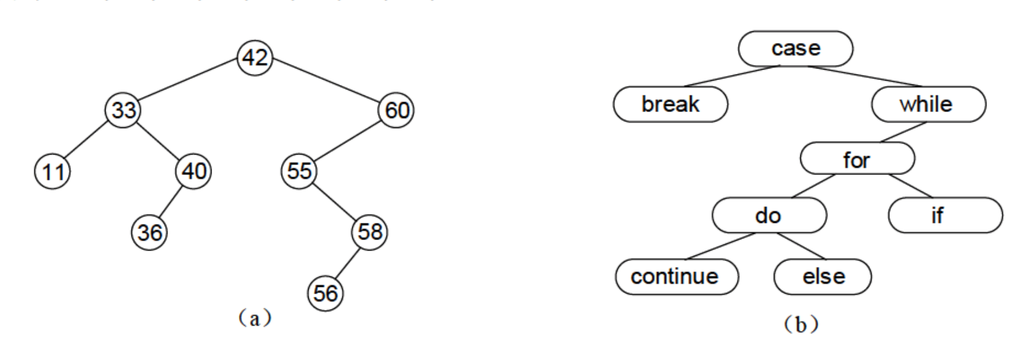


图 8.4　二叉排序树示例

不难看出，前面讨论的折半查找判定树也是一棵二叉排序树。

在下面讨论二叉排序树的操作中，使用二叉链表作为存储结构。

1．二叉排序树的插入和生成

在二叉排序树中插入新结点，只要保证插入后仍符合二叉排序树的定义即可。插入过

① 有的文献允许键值重复，比如将定义中右子树的“大于”改为“大于或等于”，则键值可重复，并依次向右子树排列。这样以后中序遍历时的重复点顺序与插入时一致，而查找时先找到的是第一个。

程是这样进行的：若二叉排序树为空，则将待插入结点*s 作为根结点插入到空树中；若二叉排序树非空，则将待插结点的关键字 s->key 和树根的关键字 t->key 比较，若 s->key=t->key，则说明树中已有此结点，无须插入；若 s->key<t->key，则将待插结点*s 插入到根的左子树中，否则将*s 插入到根的右子树中。而子树中的插入过程又和在原树中的插入过程相同，如此进行下去，直到把结点*s 作为一个新的树叶插入到二叉排序树中，或者发现树中已有结点*s 为止。

显然上述插入过程是递归定义的，容易写出相应的递归算法：

```
bitree insert(bitree t,keytype K) {//递归算法
  pointer s;
  if(t==NULL) {                    //子树为空，插入新结点
    s=new node;
    s->lchild=NULL;
    s->rchild=NULL;
    s->key=K;
    return s;
  }
  if(K==t->key) return t;          //树中已有结点*s，无须插入
  if(K<t->key) t->lchild=insert(t->lchild,K); //在左子树上插入
  else         t->rchild=insert(t->rchild,K); //在右子树上插入
  return t;
}
```

注意到插入过程是从根结点开始逐层向下寻找插入位置的，也容易写出非递归算法（略）。

以图 8.4（a）所示的二叉排序树为例，若插入关键字为 48 的结点，则插入过程如图 8.5 所示。由于插入前二叉排序树非空，故将 48 和根结点 42 比较，因 48>42，则应将 48 插入到 42 的右子树上；又因 42 的右子树不空，将 48 再和右子树的根 60 比较，因 48<60，则 48 应插入到 60 的左子树上；依此类推，直至最后因 48<55，且 55 的左子树为空，故将 48 作为 55 的左孩子插入到树中。

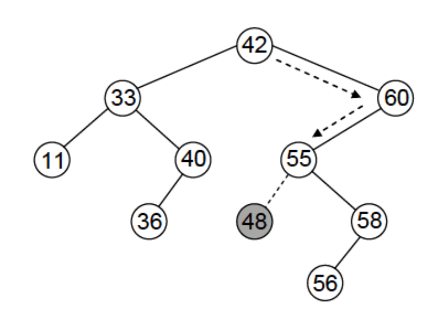


图 8.5 二叉排序树的插入

二叉排序树的生成，是从空的二叉排序树开始，每输入一个结点数据，就建立一个新结点，并插入到当前已经生成的二叉排序树中。例如，设关键字的输入次序为：12, 20, 5, 10, 9，按上述算法生成二叉排序树的过程，如图 8.6 所示。

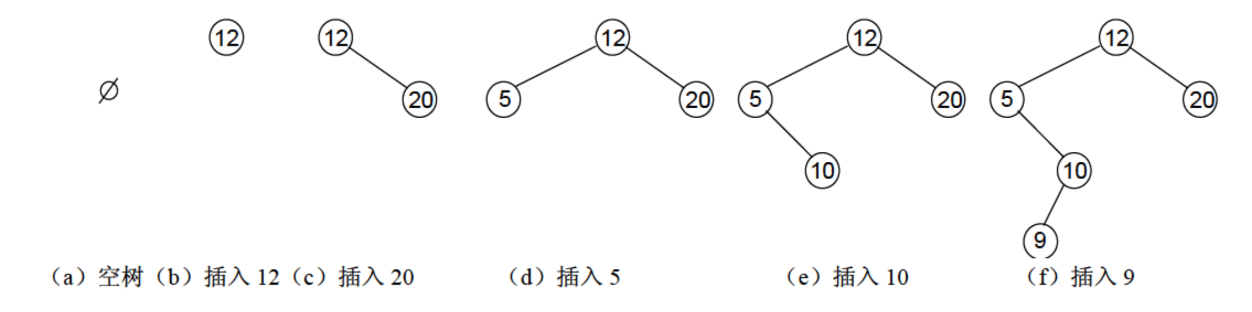


图 8.6 二叉排序树的生成过程示例

二叉排序树的生成算法如下：

```
const keytype endflag=0;          //假设输入结束标志为 0
bitree creat() {
   keytype K;
   t=NULL;                        //从空树开始
   while(cin>>K,K!=endflag)       //读入结点关键字,不是结束标志时循环
      t=insert(t,K);
   return t;
}
```

因为二叉排序的中序序列是一个有序序列，所以，对于一个任意的关键字序列构造一棵二叉排序树，其实质就是对此关键字序列进行排序，使其变为有序序列。“排序树”的名称也由此而得。

2. 二叉排序树的删除

从二叉排序树中删除一个结点，不能把以该结点为根的子树都删去，只能删掉这个结点，并且还要保证删除后所得的二叉树仍然满足二叉排序树的性质。也就是说，在二叉排序树中删去一个结点相当于删去有序序列中的一个结点。

删除操作必须首先进行查找，以确定被删结点是否在二叉排序树中。若不在，则不做任何事情；否则，设待删结点为 P，其双亲结点是 F，相应的结点指针分别为 p 和 f，不妨设 P 为 F 的左孩子（P 为 F 的右孩子时处理方法类似），分三种情况：

（1）P 为叶结点。这种情况最简单，直接删掉该结点（释放其空间），并将其双亲 F 的左指针置空，见图 8.7。

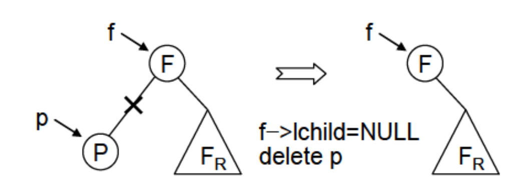


图 8.7 二叉排序树上删除叶结点

（2）P 只有一棵非空子树。该子树可能是 P 的左子树，也可能是右子树，但处理是相同的，即令该子树为 P 的双亲 F 的左子树，再删除结点 P，见图 8.8。

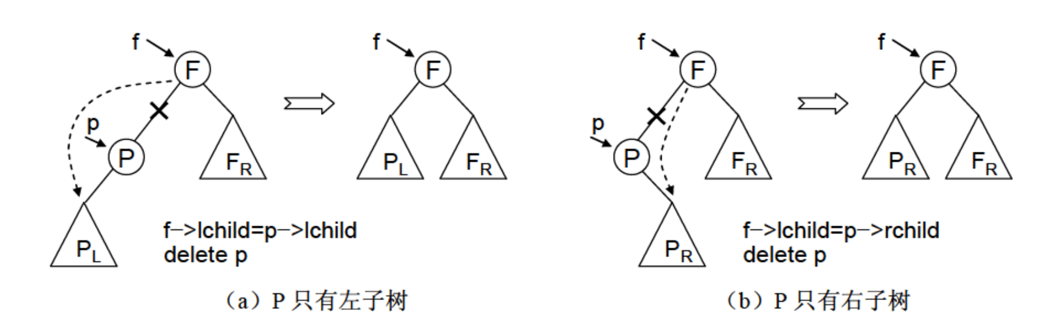
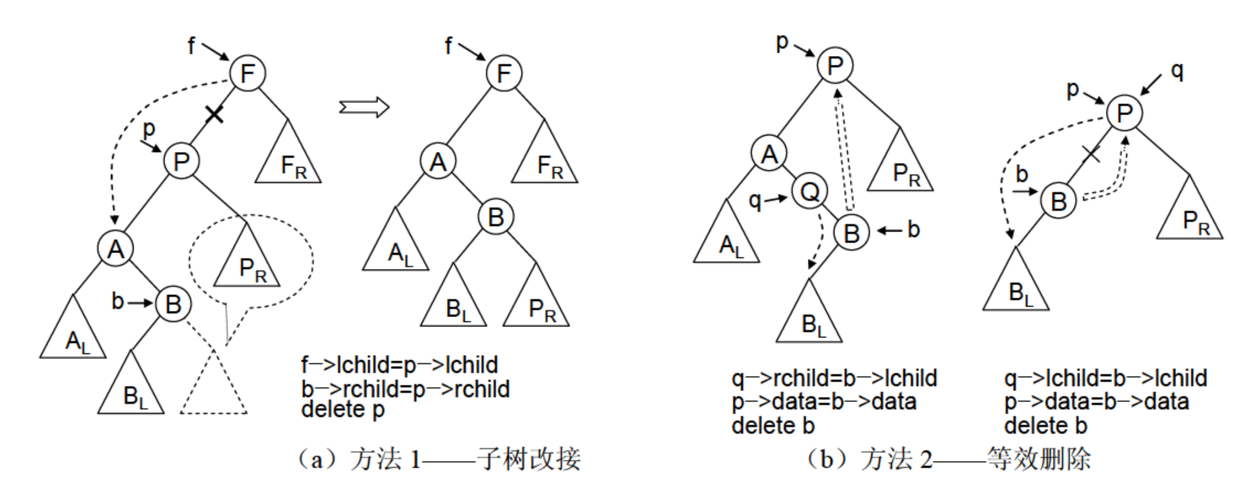


图 8.8 二叉排序树上删除只有一个子树的结点

（3）P 有两棵非空子树。这时可有多种处理方法。

方法一：子树改接。将 F 的左指针指向 P_L，将 P_R 改接到 P_L 的右下角。这需找到 P_L 最右下的结点 B（即 P 的左子树中键值最大的结点，它一定没有右孩子），让 P_R 成为 B 的右子树，见图 8.9（a）。注意，B 点也可能就是 P_L 的根 A，这时 A 没有右子树。

类似地，也可将 F 的左指针指向 P_R，将 P_L 改接到 P_R 的左下角。这时要找 P_R 最左下的结点 C（即 P 的右子树中键值最小的结点，它一定没有左孩子）。



（a）方法 1——子树改接　　（b）方法 2——等效删除

图 8.9　二叉排序树上删除有两个子树的结点

方法二：等效删除。用 P 的中序前趋 B 顶替 P，即将 B 的数据复制到 P 中，将 B 的左子树上接到其双亲结点 Q 的对应链域上，然后删去 B，见图 8.9（b）。

类似地，也可用 P 的中序后继 C 来顶替 P，将 C 的右子树上接到其双亲结点 Q 的对应链域上，然后删去 C。

显然，第一种方法可能增加树的高度，不如后一种方法好。

不难发现，情况(1)可统一到情况(2)中，见图 8.8(a)，若 P 为叶子，则 p->lchild=NULL，语句 f->lchild=p->lchild 也就是 f->lchild=NULL，即图 8.7 的处理方法。

采用上述等效删除的删除算法如下：

```
pointer del_node(pointer p) {   //p 指向待删除结点
  pointer q,b;
  if(p==NULL) return;           //空结点，不存在
  if(p->rchild==NULL) {         //P 只有左子树
    q=p; p=p->lchild; delete q;
  }
  else if(p->lchild==NULL) {    //P 只有右子树
    q=p; p=p->rchild; delete q;
  }
  else {                        //P 有 2 个子树，等效删除
    q=p; b=p->lchild;
    while(b->rchild!=NULL) {q=b; b=b->rchild;}
    if(q!=p) q->rchild=b->rchild;
    else q->lchild=b->lchild;
    p->data=b->data;
    delete b;
```

```
    }
    return p;                          //返回删除后的子树(根)
}
```

该函数的调用方法为：若待删结点 P 有双亲，假设它为双亲 F 的左孩子 f->lchild，则为 f->lchild=del_node(f->lchild)；否则 P 为根，则调用为 p=dele_node(p)。这是因为有双亲时可能要修改双亲相应的孩子指针，而无双亲时要返回删除后的结果。

3. 二叉排序树上的查找

二叉排序树可看成一个有序表，在二叉排序树上的查找过程就与二分查找有些类似，也能较快地缩小查找范围。但查找中每次不是与区间的中点作比较，而是与根比较，由此决定下一步的搜索范围是左子树还是右子树。实际上在前面介绍二叉排序树上的插入和删除操作时就已使用了查找操作。

二叉排序树上的查找算法如下：

```
pointer search(bitree t,keytype K) {    //递归算法
    if(t==NULL) return NULL;
    if(K==t->key) return t;
    if(K<t->key) return search(t->lchild,K);
    else         return search(t->rchild,K);
}
```

显然，在二叉排序树上查找，若查找成功，则是从根结点出发走了一条从根到待查结点的路径；若查找不成功，则是从根结点出发走了一条从根到某个空子树的路径（若把空子树画出来作叶子①，则也可说是走了一条从根到待查结点的路径)。因此与二分查找类似，关键字的比较次数不超过树的高度。

然而，二分查找法查找长度为 n 的有序表时，其判定树是唯一的；而含有 n 个结点的二叉排序树却不唯一。对于含有同样一组结点的表，由于结点插入的先后次序不同，所构成的二叉排序树的形态和高度也可能不同②。如图 8.10（a）所示的树，是按{12, 20, 5, 10, 9}的插入次序构成的；如果分别按{10, 5, 20, 9, 12}、{5, 9, 10, 12, 20}、{20, 5, 12, 9, 10}等插入次序则分别构成图 8.10（b）、（c）、（d）所示的树。

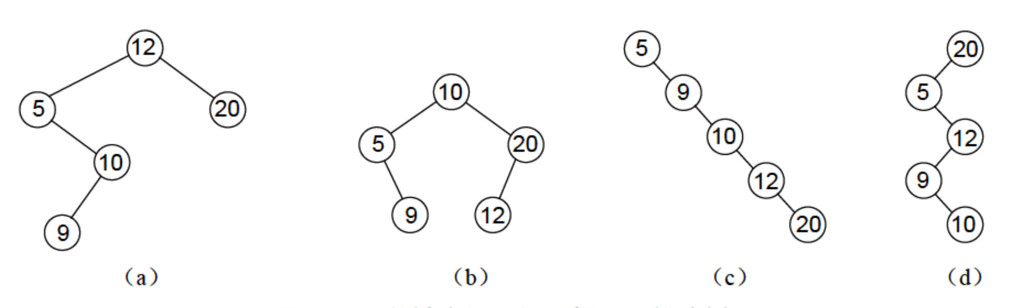


图 8.10　关键字相同的 4 棵二叉排序树

这 4 棵二叉树的高度分别是 4、3、5 和 5，因此，在查找失败的情况下，在这四棵树上所进行的关键字比较次数最多分别为 4、3、5 和 5；在查找成功的情况下，它们的平均查找长度也会不同。对子图（a），因为第 1、2、3、4 层上各有 1、2、1、1 个结点，而找到第 i 层的结点时恰好需比较 i 次，所以，在等概率假设下，查找成功的平均查找长度为：

① 这时的二叉树称为**扩充二叉树**（extended binary tree），显然为严格二叉树。

② 有些树是相同的，如{10, 5, 20, 9, 12}、{10, 5, 9, 20, 12}等。

$$ASL_a=\sum_{i=1}^{5}p_ic_i=(1+2\times2+3\times1+4\times1)/5=2.4$$

类似，在等概率假设下，子图（b）、（c）和（d）在查找成功时的平均查找长度为：

$$ASL_b=(1+2\times2+3\times2)/5=2.2$$

$$ASL_c=ASL_d=(1+2+3+4+5)/5=3$$

可见，在二叉排序树上进行查找时的平均查找长度和二叉树的形态有关。在最坏情况下，二叉树每一层只有一个结点（单枝树，相当单链表），这时高度最大，平均查找长度和单链表上的顺序查找相同，为(n+1)/2。比如，把一个有序表（递增或递减）的 n 个结点依次插入而生成的二叉排序树就蜕化为一棵高度为 n 的单枝树（右单枝树或左单枝树）。在最好情况下，二叉排序树在生成的过程中，树的形态比较匀称，最终得到的是一棵形态与二分查找的判定树相似的二叉排序树，此时它的平均查找长度大约是 $\log_2 n$ 。

若考虑把 n 个结点，按各种可能的次序插入到二叉排序树中，则有 n!棵二叉排序树（其中有的形态相同），可以证明（见习题 8.10），在等概率的情况下（即每个关键字被查概率相等），对这些二叉排序树进行平均，得到的平均查找长度约为 $1.39\log_2 n$ ，即仍然是 $O(\log_2 n)$。

这个问题也可这样理解：在二叉排序树的生成过程中，要多次进行关键字比较。每次新结点与已有结点比较后决定放到其左边或右边。如果把已有结点看成“基准”，则不难看到，这个过程与快速排序的划分实际是相同的（基准取为序列的第一个），不同的是快速排序是在所有数据已齐备时进行的，而二叉排序树是每出现一个结点就处理一下。所以利用快速排序的分析结果，马上得到：n 个结点的二叉排序树在生成过程中，关键字比较次数平均约为 $1.39n\log_2 n$ 。

就平均性能而言，二叉排序树上的查找和二分查找相差不大，并且二叉排序树上的插入和删除结点十分方便，无须移动大量结点。因此，对于需要经常做插入、删除和查找运算的表，宜采用二叉排序树结构。由此，人们也常常将二叉排序树称为二叉查找树。

若各结点的查找概率不同，则平均检索长度最小的二叉排序树称为**最优二叉排序树**。显然，查找概率高的结点应离根较近，但实际构造比较困难（通常采用近似算法）。为了保持最优，插入、删除不方便，故一般并不用于动态检索。

最后需要指出，二叉排序树一般并不直接用来排序，因为建立二叉排序树时要花费一定时间（比较次数为 $O(n^2)$～$O(n\log_2 n)$，取决于树的形态），且占用空间大（每个结点需两个附加指针），最后为了得到排序序列还要遍历，不如前一章介绍的各种排序算法好。

8.3.2 平衡二叉排序树

从上节的讨论可知，二叉排序树的查找效率取决于树的形态，而构造一棵形态匀称的二叉排序树与结点插入的次序有关。但是结点插入的先后次序往往不是随人的意志而定的，这就要求我们找到一种动态平衡的方法，对于任意给定的关键字序列都能构造一棵形态匀称的二叉排序树。

我们把形态匀称的二叉树称为**平衡二叉树**（Balanced Binary Tree），显然其中任一结点的左右子树的高度应该大致相同。一般只要二叉树的高度为 $O(\log_2 n)$就可看成是平衡的，所以平衡二叉树有多种。本节讨论的平衡二叉树是指 **AVL 树**：或者为空，或者是任何结点

的左、右子树高度相差不超过 1 的二叉树。二叉树上任一结点的左、右子树高度之差，称为该结点的**平衡因子**（Balanced Factor）[①]。因此，平衡二叉树上所有结点的平衡因子只可能是−1、0、1。换言之，若一棵二叉树上任一结点的平衡因子的绝对值都不大于 1，则为平衡二叉树。

例如，图 8.11 中子图（a）是一棵平衡二叉树，而子图（b）中有平衡因子为−2 的结点，故不是平衡二叉树，图中结点内的数字是平衡因子。

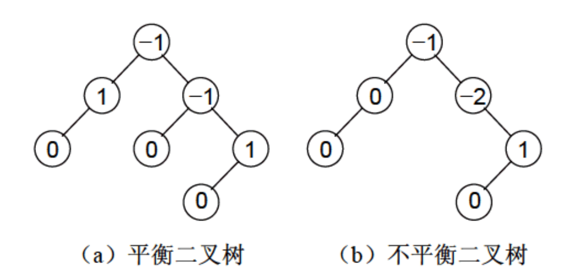


图 8.11　平衡和不平衡二叉树示例

任何一棵二叉排序树，都可以转化为一棵平衡二叉排序树。事实上，折半查找的判定树就是一棵平衡二叉树，所以，对任一棵二叉排序树，我们可将它的排序序列按折半查找判定树的生成方法，组织成一棵二叉树，它显然是二叉排序树，且是平衡二叉树。该性质的意义在于指出了平衡二叉排序树的存在性，同时也指出了它的一种构造方法，但这是一种完全重新构造的方法，不适合于动态构造。

如何构造出一棵平衡的二叉排序树呢？Adelson−Velskii 和 Landis 等人在 1962 年提出了一个动态地保持二叉排序树平衡的方法，AVL 树即因此得名。其基本思想是：在构造二叉排序树的过程中，每插入一个结点，就检查是否因插入而破坏了树的平衡性；若是，则找出其中最小不平衡子树，在保持排序树特性的前提下，调整最小不平衡子树中各结点之间的连接关系，以达到新的平衡。

所谓**最小不平衡子树**是指：以离插入结点最近、且平衡因子绝对值大于 1 的结点作根的子树。为了简化讨论，不妨假设二叉排序树的最小不平衡子树的根结点是 A（结点 B、C 的含义见相应的图示），调整该子树的规律可归纳为下列 4 种情况。

（1）LL 型调整

失衡原因：在 A 的左孩子(L)的左子树(L)上插入结点，使 A 的平衡因子由 1 变为 2（B 的平衡因子由 0 变为 1）。

调整操作："提升" B 为新子树的根；A 下降为 B 的右孩子，同时将 B 原来的右子树 B_R 调整为 A 的左子树，见图 8.12（a）所示，图中带阴影的小框表示被插入的结点。

（2）RR 型调整

失衡原因：在 A 的右孩子(R)的右子树(R)上插入结点，使 A 的平衡因子由−1 变为−2（B 的平衡因子由 0 变为−1）。

调整操作："提升" B 为新子树的根；A 下降为 B 的左孩子，同时将 B 原来的左子树 B_L 调整为 A 的右子树，见图 8.12（b）所示。

① 即左子树高度−右子树高度。有的文献定义为右子树高度−左子树高度。

（3）LR 型调整

失衡原因：在 A 的左孩子(L)的右子树(R)上插入结点，使 A 的平衡因子由 1 变为 2（B 的平衡因子由 0 变为−1）。

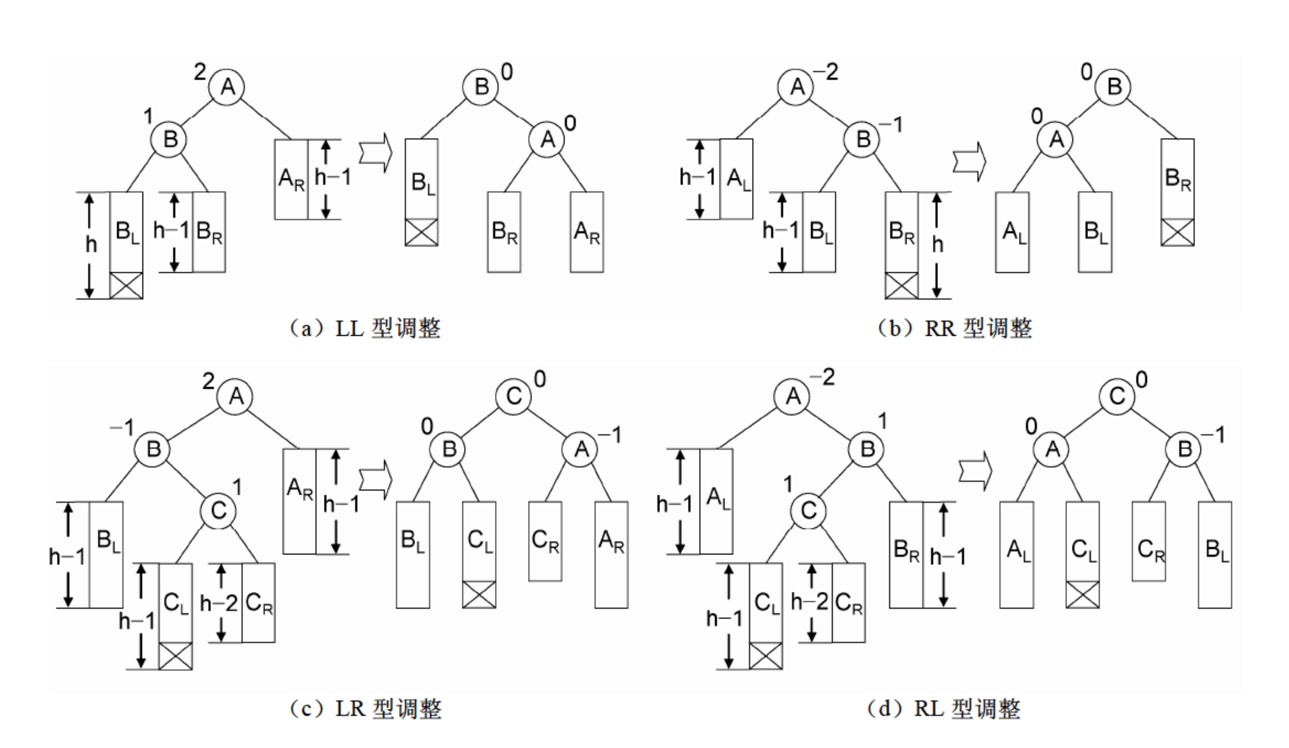
调整操作：“提升”C 为新子树的根；A 下降为 C 的右孩子（B 变为 C 的左孩子），同时将 C 原来的左子树 C_L 调整为 B 的右子树，C 原来的右子树 C_R 调整为 A 的左子树，见图 8.12（c）所示（插入点也可能在 C_R 中，此时 C 的平衡因子为−1；插入点也可能就是 C，此时 C 为新结点，C_L、C_R 为空；但调整操作相同）。

（4）RL 型调整

失衡原因：在 A 的右孩子(R)的左子树(L)上插入结点，使 A 的平衡因子由−1 变为−2（B 的平衡因子由 0 变为 1）。

调整操作：“提升”C 为新子树的根；A 下降为 C 的左孩子（B 变为 C 的右孩子），同时将 C 原来的左子树 C_L 调整为 A 的右子树，C 原来的右子树 C_R 调整为 B 的左子树，见图 8.12（d）所示（插入点也可能在 C_R 中，此时 C 的平衡因子为−1；插入点也可能就是 C，此时 C 为新结点，C_L、C_R 为空；但调整操作相同）。

易见，这四种情况实际上只有两种是独立的，因为 RR 与 LL 对称、RL 与 LR 对称。



（a）LL 型调整　（b）RR 型调整

（c）LR 型调整　（d）RL 型调整

图 8.12　AVL 树的四种平衡调整示意图

在上述的调整操作中，仅需改变少量的指针，而且调整后新子树的高度和插入前子树的高度一样，因此，无须考虑变动最小不平衡子树之外的结点，就可完成对整个二叉排序树的平衡。另外，调整前后叶子结点从左到右的相对次序保持不变。至于调整的正确性，可简单地检验中序序列是否仍然递增有序即可。

下面简单介绍一下 AVL 树的插入算法。显然，AVL 树的插入算法应在二叉排序树插入算法基础上扩充以下功能：

（1）判断插入结点后是否失衡。

（2）若是，寻找最小失衡子树并转（3）。

（3）判断失衡类型并做相应调整。

易知，失衡的判断可以与寻找最小失衡子树结合起来（一棵 AVL 树失衡当且仅当它有失衡子树）。而一棵子树是否失衡可由它根结点的平衡因子的绝对值是否大于 1 决定。进一步，如果失衡，最小失衡子树的根一定是离插入结点最近且插入前平衡因子的绝对值为 1（因而插入后才可能大于 1）的结点。这样，寻找最小失衡子树的过程可以进一步与寻找新结点的插入位置的过程结合起来。AVL 树插入算法的基本步骤如下：

（1）在寻找新结点的插入位置的过程中，记下离该位置最近且平衡因子不等于 0 的结点 A，它是可能出现的最小失衡子树的根。

（2）修改自该结点到插入位置路径上所有结点的平衡因子，其他结点的平衡因子不受影响。

（3）判断插入结点后，结点 A 的平衡因子的绝对值是否大于 1，若是，进一步判断失衡类型并作相应的调整；否则本次插入过程结束。

算法中要用到结点的平衡因子，可在二叉树的每个结点中增加一个平衡因子域，具体算法这里从略。

易见，高度为 h 的 AVL 树结点数最少时，其两个子树的高度分别为 h−1 和 h−2，且子树的结点数也最少，于是 $n_h=n_{h-1}+n_{h-2}+1$，即$(n_h+1)=(n_{h-1}+1)+(n_{h-2}+1)$，所以最少结点数 n_h+1 与对应的 Fibonacci 数相同：$n_h+1=F_{h+2}\approx\frac{1}{\sqrt{5}}\phi^{h+2}$ $(\phi=\frac{1+\sqrt{5}}{2},\ h\geqslant 0)$（见附录 E）。据此可知：含有 n 个结点的 AVL 树，树的高度最大约为 $\log_\phi(\sqrt{5}(n+1))-2\approx 1.44\log_2 n$。实际 AVL 树的结点分布基本不会如此稀疏，高度多数是接近理想平衡时的 $\log_2 n$。可以证明，平均高度为 $O(\log_2 n)$。

在 AVL 树上查找时，和关键字比较的次数不会超过树的高度，且不会出现蜕变为单枝树的情形，因此，查找 AVL 树的时间复杂度是 $O(\log_2 n)$。然而，动态平衡过程需花费不少时间，故在实际应用中是否采用 AVL 树，还要根据具体情况而定。一般情况下，若结点关键字是随机分布的，并且系统对平均查找长度没有苛求，则不必使用平衡二叉排序树。

与插入操作类似，在删除操作时 AVL 树也可能失去平衡而需要调整，但每次插入时最小不平衡子树调整后树的其他部分不受影响（仍平衡），故最多调整一次；而每次删除时当前最小不平衡子树调整后可能沿根的方向产生新的最小不平衡子树，又要调整，即可能要从最初的最小不平衡子树向根调整多次，但最多 $O(\log_2 n)$次，具体略。

8.3.3　B 树[①]

至此，我们讨论的查找算法都是内查找算法，这是因为被查找的数据都是保存在内存中的。它们适用于较小的查找表，而对较大的、存放在外存储器上的文件就不合适了。例如，当用平衡二叉树作为磁盘文件的索引组织时，若以结点作为内、外存交换的单位，则

① 有文献写做 B¯树、B_树、B-树、B−树等（不能读做“B 减树”，英文为 B-tree，中间为连字符）。

查找到需要的关键字之前，平均要对磁盘进行 $\log_2 n$ 次访问，这是很费时间的。

为了减少访外次数，需要降低查找树的高度，这可采用多叉树，特别是平衡多叉树。为此，1972 年，R.Bayer 和 E.M.McCreight 提出了一种适用于外查找的 B 树，它是一种平衡多叉树，每个结点存放多个关键字。B 树常用作索引，在文件系统和数据库系统中获得过重要应用。

1．B 树的定义

一棵 m 阶的 B 树，或者为空树，或者为满足下列条件的 m 叉树：

（1）根至少两个孩子。

（2）除根之外，每个内部结点至少 $\lceil m/2 \rceil$ 个孩子（最多 m 个孩子）。

（3）除叶子外，每个结点的关键字从小到大排列，孩子数比关键字个数多 1。

（4）所有叶子在同一层，叶子不含任何关键字信息（实际为树中并不存在的外部结点，且指向这些外部结点的指针为空）。

以上条件（1）实际上是多余的，因为只要（内部）结点存在，它必含关键字，而即使一个关键字也有两个孩子。如果 B 树的阶为 2，则每个结点最多和最少都是两个孩子，于是每个结点都有两个孩子，而叶子在同一层，使得 2 阶 B 树只能是满二叉树。3 阶 B 树也称为 **2～3 树**，因为每个内部结点有 2 个或 3 个孩子。

条件（1）也使 B 树不至于一开始就偏向一边；条件（2）使每个结点至少**半满**；条件（4）中叶子都在同一层，使 B 树高度上平衡；条件（3）中结点关键字递增排列，使 B 树有某种“中序”递增性，可看成二叉排序树的扩充，是一种平衡多叉排序树。由于 N 个关键字查找失败的情况有 N+1 种（分别对应关键字之间的区间），所以 B 树中叶子数比树中全部关键字个数多 1。

例如，图 8.13 所示的树是一棵 m=5 阶的 B 树，其高度为 4。叶结点用圆圈表示，不含任何信息，都在第 4 层。其他结点用矩形表示，里面的数字为关键字。根结点有两个孩子，包含一个关键字。其他非叶结点的孩子个数在 $\lceil m/2 \rceil$=3 到 5 之间，相应地，结点所包含的关键字个数在 2 到 4 之间。在每个非叶结点中，关键字是按递增顺序排列的，且（孩子）指针的数目比该结点的关键字个数多 1 个。

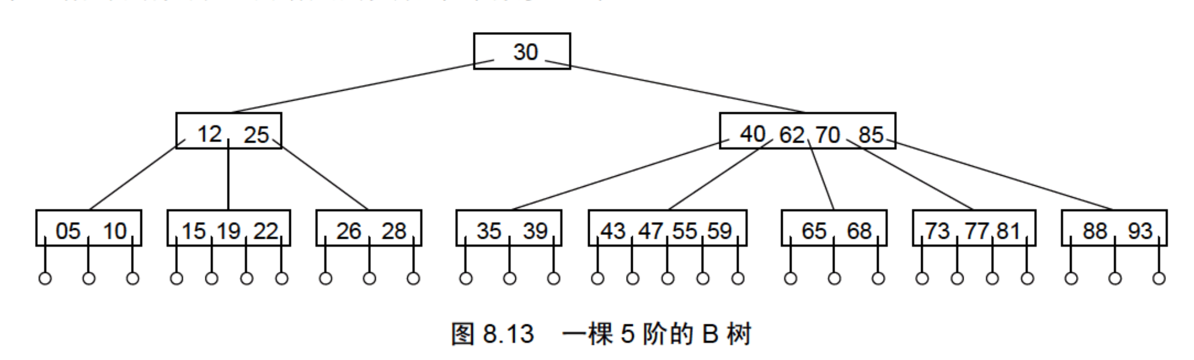


图 8.13　一棵 5 阶的 B 树

在 B 树中包含 j 个关键字、j+1 个指针的结点，一般形式为($P_0, K_1, P_1, K_2, \cdots, P_{j-1}, K_j, P_j$)。其中，$K_i$ 是关键字，且 $K_1<K_2<\cdots<K_j$；p_i 是指针，指向包含 K_i 到 K_{i+1} 之间的关键字的子树。例如图 8.13 所示的 B 树，根结点有两个指针，一个指向包含小于 30 的那些关键字的子树，另一个指向包含大于 30 的那些关键字的子树。

实际上，结点中每个关键字本身还需要一个指向其所在记录位置的指针。另外，在下面将看到，在 B 树中插入和删除时可能要向上访问，故结点中一般还要增加一个双亲指针。在操作中，还可使 B 树结点的大小填满一个磁盘块，这样结点中的（孩子）指针就是孩子结点所在的块号。

2．B 树的运算

（1）查找

在 B 树中查找给定关键字的方法是：从根结点开始，在根结点所包含的关键字 K_1、K_2、…、K_j 中查找给定的关键字，若找到则查找成功；否则，一定可以确定要查的关键字是在某个 K_i 和 K_{i+1} 之间（因为结点内的关键字是有序的），于是，取 p_i 所指向的子树继续查找。如此重复下去，直到查找成功或指针 p_i 为空（对应叶子结点）时，查找失败。

显然，若每个结点只有一个关键字，则查找过程与二叉排序树基本相同。在每个结点中找关键字时，由于各关键字是有序的，既可顺序查找，也可二分查找。

上述查找过程包含两种基本操作：在 B 树上找结点；在结点中找关键字。由于 B 树通常存储在磁盘上，则前一查找是在磁盘上进行的，读入结点的信息后，后一查找在内存中进行。由于访外（磁盘）比较耗时，所以，在 B 树上找结点的效率可用来衡量 B 树的查找效率。在 B 树上找结点时所需比较的次数就是该结点在 B 树中的层数。

以最坏情况为例，即待查结点在 B 树的最低层 L 上（非叶子层），我们来看看 B 树的查找效率。设 m 阶 B 树包含 N 个关键字，则第 L+1 层有 N+l 个叶结点。第一层为根，至少一个结点；根至少有两个孩子，故第二层至少两个结点。除根和叶结点之外，其他结点至少有 $\lceil m/2 \rceil$ 个孩子，因此，第三层至少有 $2\times\lceil m/2 \rceil$ 个结点，第四层至少有 $2\times\lceil m/2 \rceil^2$ 个结点，……，那么第 L+1 层至少有 $2\times\lceil m/2 \rceil^{L-1}$ 个结点，于是有 $N+l \geqslant 2\times\lceil m/2 \rceil^{L-1}$，即：

$$L \leqslant 1+\log_{\lceil m/2 \rceil}((N+1)/2)$$

这也是 B 树的最大高度。这个结果比平衡二叉排序树的 $\log_2 n$ 好得多，因为一般 $\lceil m/2 \rceil >> 2$。只要阶数 m 足够高，查找 B 树结点的次数是可以很低的，如以 m=199 为例，即使 N=1 999 999，则 L 至多等于 4。由于对 B 树的一次查找至多进行 L 次存取，这个公式保证了 B 树的高效率查找。

类似可知 B 树的最小高度 $L \geqslant \log_m(N+1)$。

（2）插入

在 B 树中插入关键字时，要先进行查找，若树中已有该关键字则不再插入；若是一个新关键字，则查找一直进行到最底层也找不到。设叶结点为第 L+1 层，则新插入的关键字总是进入第 L 层的结点。这与二叉排序树不同，后者插入的结点可分布在各层。

设 B 树的阶为 m，则每个结点关键字的个数最多为 m−1 个。若要插入的结点关键字个数少于 m−1 个，则插入过程仅局限于该结点，只要把新关键字直接插入该结点即可。

若要插入的结点已有 m−1 个关键字，则插入后关键字的个数将超过允许的最多个数 m−1，这时要将该结点**分裂**为两个，并把中间关键字提升到双亲结点里去（使分裂后的两个结点大小相当，都约半满）。如果双亲结点原来也是满的，就需要继续分裂和提升。最坏情况是这个过程一直传播到根。若根也需要分裂，由于它没有双亲，则要另外建立一个新的根结点，整个 B 树就增加了一层。

例如，在图 8.13 的 5 阶 B 树中插入关键字 60，被插入结点的关键字个数原来就有最多允许的 4 个，插入后就会有 5 个：(43, 47, 55, 59, 60)，对该结点分裂，提升中间关键字 55 到双亲结点，结果双亲结点又要分裂，最后结果如图 8.14 所示[①]。

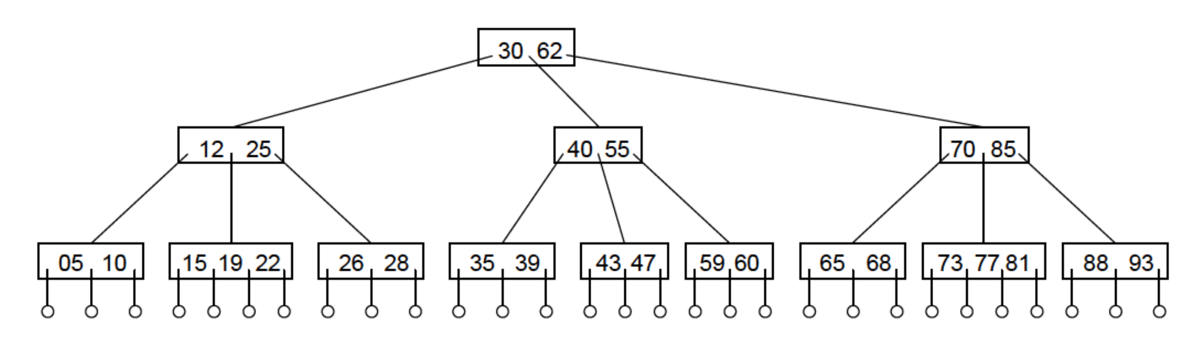


图 8.14　在图 8.13 上插入关键字 60 引起结点分裂

可见，B 树的插入及其调整是从下向上进行的。

如果通过插入的方法生成 B 树，则 B 树将是从底部向上生长的。

（3）删除

删除关键字时也要先进行查找。若删除的关键字不在第 L 层，则先把该关键字与它在第 L 层的后继（或前趋）交换位置（注意后继、前趋总在第 L 层），然后在第 L 层中删除该关键字。这样便于删除后对结点从下向上调整（与插入类似）。例如，在图 8.14 的 B 树中删除关键字 70，它不在最底层，则先找到 70 的后继 73，交换两者位置，然后再删除 70，结果如图 8.15（a）所示。

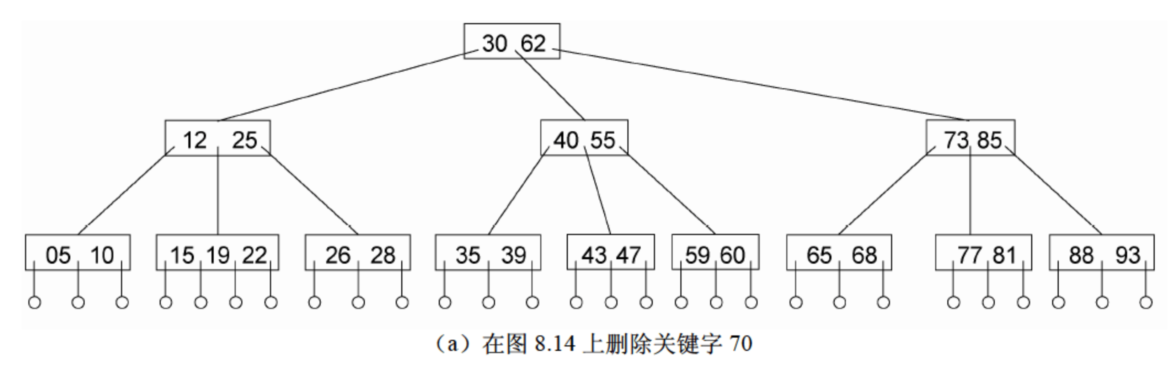
删除第 L 层的关键字后，可能导致所在结点的关键字个数少于$\lceil m/2 \rceil-1$（不足半满）。这时要在该结点左或右兄弟结点中**移动**若干个关键字到该结点中来（其中最小或最大的关键字要提升到双亲结点，双亲结点相应的一个关键字要移动下来，这相当于移动是通过双亲绕行的），使两个结点所含关键字个数基本相同。若兄弟结点的关键字个数也很少，刚好等于$\lceil m/2 \rceil-1$，就不能进行这种移动，这时要把该结点、它的兄弟结点以及它们双亲结点中相应的一个关键字**合并**为一个结点。但从双亲结点中拿走一个关键字后，双亲结点本身可能又要合并，有时这种合并可一直传播到根结点；特别地，如果传播到根而根结点只包含一个关键字，则根结点关键字下移与它的两个孩子合并后，形成的是新的根结点，整个树就减少了一层。

例如，从图 8.15（a）中删除 10，这使原包含关键字 10 的结点只剩下一个关键字，即小于半满所需的$\lceil m/2 \rceil-1=\lceil 5/2 \rceil-1=2$。于是需要从右边兄弟结点移一个关键字 15 到该结点来，但这又涉及它们双亲结点中的关键字 12 也要作相应变化，所以，实际上是将右兄弟中关键字 15 上移到双亲结点，而把双亲中关键字 12 下移到原来包含 10 的结点，如图 8.15（b）所示。

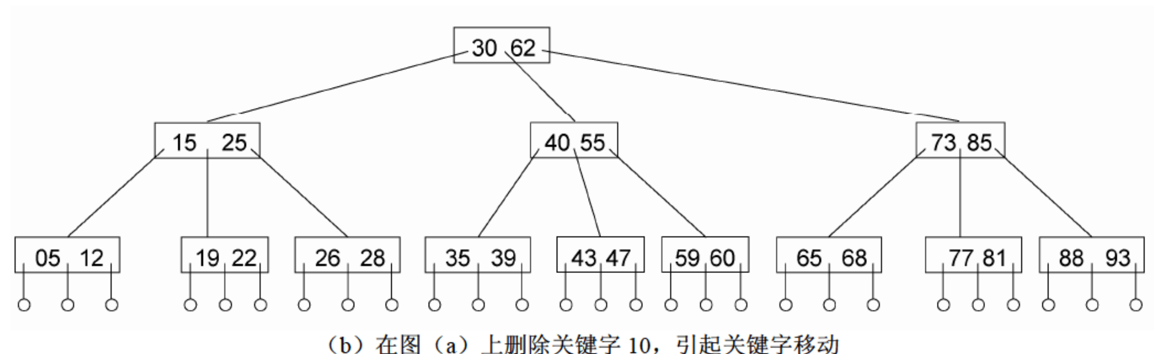
如果在图 8.15（b）中再删除关键字 19，则删除 19 后，原包含 19 的结点只剩下一个关键字 22，并且它的左、右兄弟结点包含的关键字也很少，刚好等于半满$\lceil m/2 \rceil-1=\lceil 5/2 \rceil-1=2$，于是，把原包含 22 的结点、它的右兄弟结点及它们双亲结点中的关

① 该图是 4 阶的，小于原来的 5 阶。也即在插入、删除等运算后，B 树的阶数可能（临时性）变小。

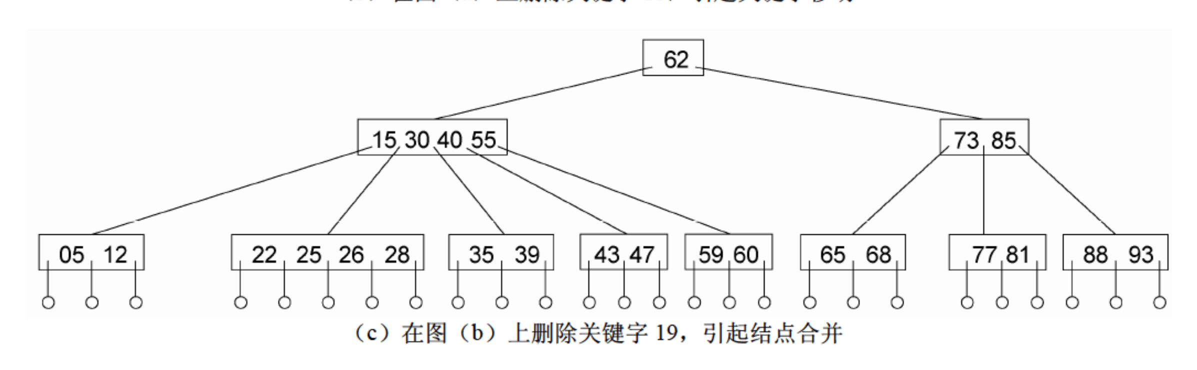
键字 25 合并成一个新结点。从双亲结点中拿出一个关键字后，双亲结点本身又要进行合并，最后结果如图 8.15（c）所示。



（a）在图 8.14 上删除关键字 70



（b）在图（a）上删除关键字 10，引起关键字移动



（c）在图（b）上删除关键字 19，引起结点合并

图 8.15　删除关键字时 B 树结点的几种变化情况

8.3.4　B^+树

上一节介绍的 B 树在文件系统上的应用实际上并不多，普遍使用的是它的一个称为 B^+树的变型树。一棵 m 阶的 B^+树和 m 阶的 B 树的差异是[①]：

（1）有 k 个孩子的结点必有 k 个关键字（叶结点的孩子是外部结点）。

（2）上面各层结点中的关键字，均是下一层相应结点中最大（或最小）关键字的复写，即上层结点是下层结点的索引。

（3）所有关键字均出现在叶结点上，叶结点包含了全部关键字的信息及指向相应记录

① 有的文献定义了另一种 B+树：除了包含全部关键字的叶子层外，上面各层的索引结构同前述 B 树。但这样定义的 B+树有的文献也称做 B 树。

的指针，且叶子结点本身依照关键字的大小，按从小到大的顺序链接。

图 8.16 是一棵 3 阶 B^+树，注意其高度为 3。

通常 B^+树上有两个头指针，一个指向根结点，另一个指向关键字最小的叶结点。因此，可以对 B^+树进行两种查找运算：一种是从最小关键字起顺序查找；另一种是从根结点开始进行随机查找。

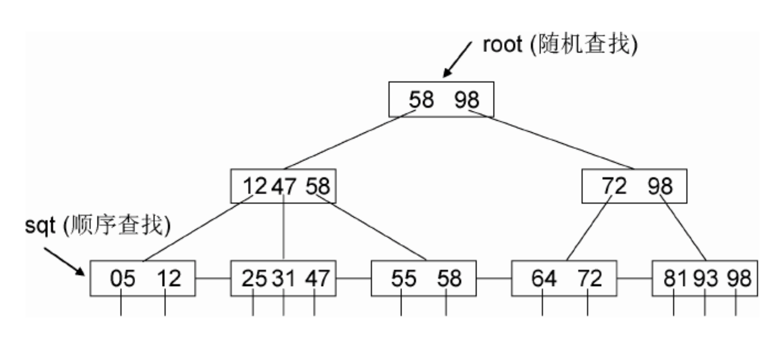


图 8.16 一棵 3 阶的 B^+树

与 B 树类似，B^+树的构造也是由下而上进行的，m 限定了结点的大小。

在 B^+树上进行随机查找、插入和删除的过程，基本上与 B 树类似。只是在查找时，若非终端结点上的键值等于给定值，并不终止查找过程，而是继续向下查找直至叶结点。因此，在 B^+树中，不管查找成功与否，每次查找都是走了一条从根到叶结点的路径。

B^+树的插入仅在叶结点上进行。当结点中的关键字个数大于 m 时要分裂成两个结点，它们所含关键字的个数分别为$\lceil(m+1)/2\rceil$和$\lfloor(m+1)/2\rfloor$，并且，它们的双亲结点中应同时包含这两个结点的最大关键字。

B^+树的删除也仅在叶结点上进行。当叶结点中的最大关键字被删除时，其在非终端结点中的值可以作为一个“分界关键字”存在（即不必修改为新值）。若因删除而使结点中关键字的个数少于$\lceil m/2\rceil$时，则要和该结点的兄弟结点进行合并。

B^+树广泛地使用在包括 VSAM 文件在内的多种文件系统中，其中每个叶子的关键字一般不是对应一个记录（稠密索引），而是对应一个页块（稀疏索引）。也正是由于使用的广泛性，对其改进也有较大的意义。一种改进方法是 B^*树，它是 B^+树的变体，除了分裂和合并结点的规则不同外，二者完全相同。B^*树在结点关键字过多时，并不马上分裂，而是先将一些记录分给相邻的兄弟结点。如果兄弟结点也满了，就将这 2 个结点分裂成 3 个。同样，当一个结点的关键字不足时，就将它与两个兄弟结点合并，使 3 个结点减少为 2 个。其目的是使结点内维持较多的关键字，提高结点存储密度，并提高检索效率。这个思想还可继续推广，使更多的结点参与合并和分裂，但算法也会更加复杂些。

以上介绍的二叉排序树、AVL 树、B 树和 B^+树等，结点内存放着完整的关键字。若关键字位数较多（或由若干部分组合而成），还可把关键字按位（或组成部分）分解后分别存放到从根到该结点的路径上，这可使关键字中相同的前缀部分共用储存空间。这类树称为**键树**。比如，若图 5.26 所示的树为键树，则叶子 I 表示串“ACFI”，叶子 G 表示串“ACG”等。实际上图 5.36 所示的哈夫曼编码树就是一种键树，如叶子 c 对应编码“110”，叶子 d 对应编码“1110”等。

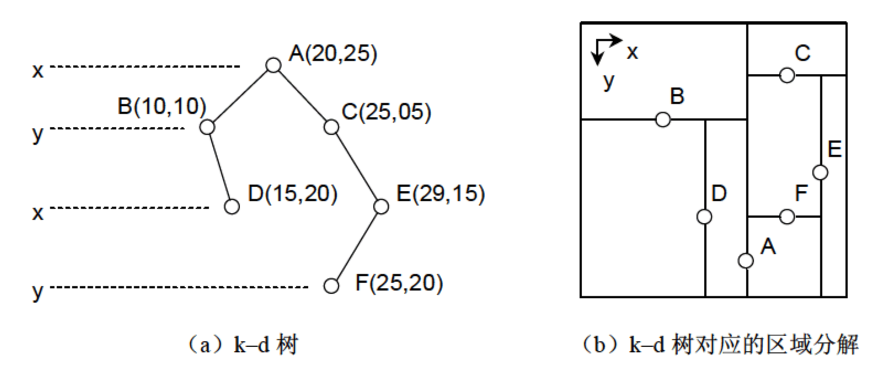
8.3.5* 空间树表

前面我们讨论的二叉排序树、B 树等都只能用于单关键字查找。如果要进行多关键字查找，一个方法是将多关键字合成为一个关键字，再采用前述的各种查找方法。但这样一来就不适合进行范围查找了；另一个方法是分别对每个关键字建立一个查找表，查找时在相应的表中进行。但多关键字查找较好的方法是采用**空间数据结构**（Spatial Data Structure），因为每个关键字相当于多维空间中的一维。这方面的应用也很多，如地理信息系统、计算机图形学等对于物体位置的查找和管理等。本节简单介绍这方面的几个数据结构。

1. k-d 树

k-d 树是对二叉排序树的改进，它能有效处理多关键字查找问题。由于二叉排序树的每个结点只能根据一个关键字产生分支，所以对多关键字问题，每层结点交替地根据其中的一个关键字产生分支。具体就是，如果关键字有 k 个，则第 1, 2, …, k, k+1, k+2, …层的结点分别根据第 1, 2, …, k, 1, 2, …个关键字产生分支。一般地，设某结点当前层数为 i，则该结点分支时所根据的关键字序号为(i−1)%k+1。

例如，对二维空间的一组数据点 A(20, 25)、B(10, 10)、C(25, 05)、D(15, 20)、E(29, 15)、F(25, 20)，相应的 k-d 树如图 8.17（a）所示。这里每个结点有两个关键字，即该结点的 x，y 坐标。第一个点 A 即为根；第二个点 B 与根 A 比较，A 点为第一层，按第一个关键字 x 分支，而 B 的 x 坐标比 A 小，所以应插入到 A 的左子树中，但 A 左子树为空，结果 B 成为 A 的左孩子；类似，第三个点 C 的 x 坐标比 A 大，成为 A 的右孩子。第四个点 D 的 x 坐标比 A 小，应插入到 A 的左子树中，此时左子树非空，则继续与左树的根 B 比较，B 点为第二层，按第二个关键字 y 分支，所以 D 应插入到 B 的右子树中，结果成为 B 的右孩子。其他点的分析类似。



（a）k-d 树　　（b）k-d 树对应的区域分解

图 8.17　k-d 树示例

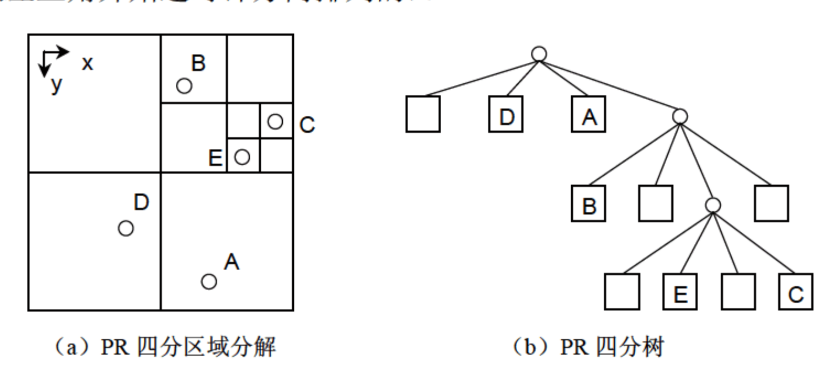
上述过程实际上就是 k-d 树的插入和生成过程。其中寻找插入位置的过程也就是 k-d 树的查找过程。这些操作与二叉排序树是类似的，只是要交替改变比较规则。k-d 树的删除过程也与二叉排序树类似，设待删除的点为 N，则一般方法是用 N 的右子树中分支规则与 N 相同的最小值结点替代 N，或者用 N 的左子树中分支规则与 N 相同的最大值结点替代 N。但这里最小值或最大值结点就不一定是相应子树中最左下或最右下的结点了，需要进行查找。

对上述空间数据点，结点的每次分支实际上相当于将该点所在的区域分成两部分，所以数据点所在的整个区域就相当于分别按（x, y）分割成了若干矩形，见图 8.17（b）所示。注意其中的坐标系为左手系。

2．PR 四分树

点-区域四分树（Point-Region Quadtree，PR 四分树）是一种严格四叉树，其结点要么是叶子，要么有四个孩子。PR 四分树主要用于二维区域的分解：设二维区域有若干数据点，现要将该区域划分成若干小区域，使每个小区域中最多只包含一个数据点。不妨设原始区域为矩形，这个要求可以这样完成：首先将区域一分为四，如果其中某个子区域的数据点多于一个，继续对该区域一分为四。最后，所有子区域要么没有数据点，要么只含有一个数据点。这个过程可以很方便地用一个四叉树来表示。

以每次四等分为例，图 8.18 表示了一个矩形区域分解的结果和相应的 PR 四分树。其中每个分支结点对应一次四分过程，这些点上不存放数据；所有数据点存放在叶子中（但有些叶子不含数据点，对应空的子区域）。注意，各个分支结点的四个子树的排列顺序要相同（图中是从左上角开始逆时针方向排列的）。



（a）PR 四分区域分解　　（b）PR 四分树

图 8.18　PR 四分树示例

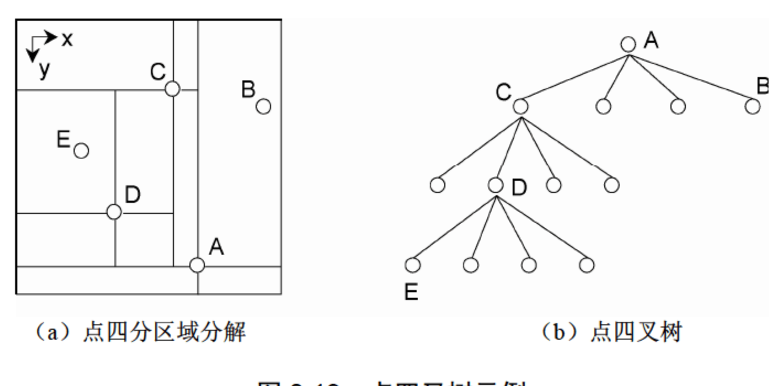
查找时从根结点开始，不断进入待查点所在的子区域，直到叶子区域，然后检查该叶子的内容是否是所求的数据点。插入数据点 x 时，先进行查找，直到某个叶子，如果该叶子不含数据点，就将 x 插入到该叶子；如果该叶子已含数据点 x，则报告数据点重复，否则将该叶子区域一分为四，最后将 x 插入到其中一个子区域。删除时也是先进行查找，直到某个叶子，如果该叶子就是待删点，则将其内容清空。但这时删除操作并不马上结束，而是检查该叶子的三个兄弟结点，如果它们也为空或只有一个数据点，则将这四个小区域合并为一个大区域，它对应一个新的叶子；若该叶子和它的兄弟结点中又都为空或只含一个数据点，则继续合并下去。

显然，对于三维空间，与 PR 四分树对应的是 PR 八分树（Point-Region Octree），即每次将区域分成八个小区域，一般也采用八等分的形式。

另外，如果在每次区域分解时以数据点为中心将区域一分为四，则对应的四叉树称为**点四叉树**（Point Quadtree），如图 8.19 所示。在这种四叉树中，分支结点和叶结点都可能存放区域的数据点信息。

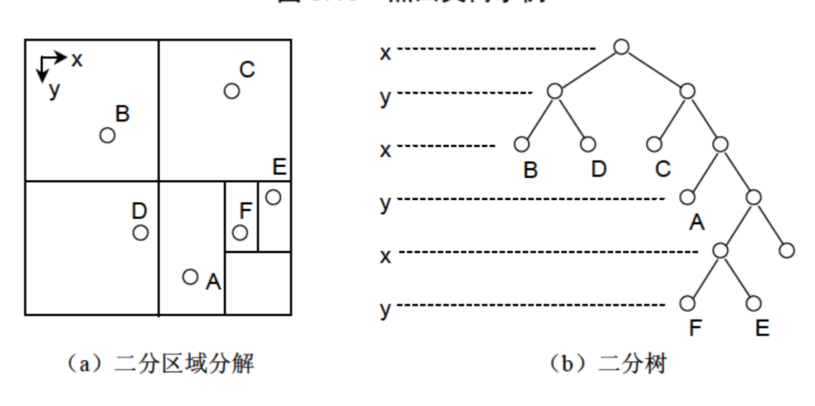
还可将 k-d 树和 PR 四分树结合起来，交替地根据坐标分量将区域一分为二，对应的

二叉树称为**二分树**（Bintree），其中每个分支结点对应一次二分过程，如图 8.20 所示。



（a）点四分区域分解　　（b）点四叉树

图 8.19　点四叉树示例



（a）二分区域分解　　（b）二分树

图 8.20　二分树示例

与 PR 四分树类似，二分树的分支结点不存放数据，所有数据点都存放在叶子中（但有些叶子不含数据点，对应空的子区域）。因为每次分解出的小区域数少，需要的分解次数多，所以二分树的高度一般比 PR 四分树大。不难理解，二分树就是二分区域分解的判定树。

8.4　散列表

在前面介绍的静态查找表和用于动态查找的树表中，结点的存储位置和结点的关键字之间不存在确定的关系，要查找某个结点需要进行一系列的关键字比较。这类查找方法建立在“比较”的基础上，每次比较后可缩小查找范围，查找效率依赖于查找过程中进行比较的次数。是否可以不作比较就可以得到记录的存储地址，从而找到所要的结点呢？回答是肯定的，这就是散列技术。

8.4.1　散列表的基本概念

散列（Hashing）既是一种储存方式，又是一种查找方法，其基本思想是根据键值直接访问表。一般过程如下：以结点的关键字 key 为自变量，通过一个确定的函数关系 f，计

算出相应的函数值 f(key)，把这个值解释为结点的储存地址，将结点存入该位置。查找时根据要查找的关键字用同样的函数计算地址，然后到相应的单元里去取值。因此，散列法又称关键字-地址转换法。上述函数 f 称为**散列函数**，函数值 f(K)称为**散列地址**，根据关键字将记录映射到表中的过程称为散列，按散列方式构造的储存结构称为**散列表**（Hash Table）。

下面看一个简单的例子。

例 8.1 已知一个含有 80 个结点的散列表，其关键字都是两位十进制的数字（关键字互不相同），则可将关键字为 i 的结点存在数组 HT[100]的第 i 号元素中，即取散列函数为：

```
H(key)=key
```

在理想情况下，散列函数是个一一对应的单调函数，即不同的键值对应不同的散列地址，比如上例。但在实际应用中，散列函数通常是个多对一的函数（主要原因是关键字取值集合大，地址集合空间小，前者到后者是一个压缩映像），于是，经常出现不同的键值对应相同的散列地址。这种现象称为**冲突**（Collision），发生冲突的关键字称为**同义词**（Synonym）。极端情况是所有关键字都是同义词，这时散列表就完全失去了意义。

对一些特定情况，有可能设计出完全不发生冲突的散列函数，称之为**完美散列**（Perfect Hashing），但代价会很高，除非确有必要，如记录集不变但要大量使用的情况，一个典型例子是对程序语言中的保留字进行散列。一般而言，冲突是不可避免的，只能尽量减少冲突。一旦发生了冲突，就必须采取适当的方法进行处理（将冲突项放到合适位置）。因此，采用散列技术时需要解决的两个主要问题是：散列函数的构造和冲突的处理。

例 8.2 已知散列表的关键字集合为：

```
S={do,for,while,continue,break,if,else,goto}
```

注意到关键字的首字母不同，可将每个结点存储到由 26 个单元组成的数组中，结点在数组中的位置，即数组下标取为关键字的首字符在 26 个英文字母表{a, b, …, z}中的序号（这里取序号从 0 开始，序号范围是 0～25），即取散列函数为：

```
H(key)=key[0]-'a'
```

每个数组单元要存放一个字符串，故整个数组可取为 char HT[26][9]。

如果再增加 2 个关键字 fopen、fprintf，则在上述散列函数下必定出现冲突，因为有 3 个关键字的首字符相同，都为“f”，于是散列地址也相同。这时可取散列函数 H(key)为 key 中首尾字母在字母表中序号的平均值，由于尾字母不同，散列地址就不再冲突了。

根据散列表的基本原理可知，它不适合按键值的大小顺序或访问的频率顺序来存放或查找结点；不适合范围检索，也不适合查找最大或最小键值的结点，只适合检索指定键值的结点。

散列表的逻辑结构是集合。按组织形式的不同，通常有两类散列表：闭散列表和开散列表。从形式上看，它们相当于一般数据结构常用的顺序存储方式和链式存储方式，但我们并不称它们为“顺序散列表”和“链式散列表”，主要是含义和结果不同。这里结点的地址不反映逻辑关系，但与其内容或关键字有关，还与冲突处理的方法有关。另外，散列表中一般不允许有键值相同的结点。

散列表的存储空间一般是个一维数组，散列地址是数组的下标。在不至于混淆时，我

们也将这个一维数组空间简称为散列表。数组空间的大小称为表的容量或表长。设散列表的长度为 m，填入表中的结点数是 n，则称 α=n/m 为散列表的**装填因子**（Load Factor）。如对上面例 8.1，装填因子为 80/100=0.8。

8.4.2　散列函数的构造方法

散列函数的种类很多，一般构造或选取散列函数的基本原则是简单和均匀。前者指散列函数的计算简单快捷，后者指散列函数能把记录以相同的概率分布到散列表的任何位置，以尽量减少冲突。如果关键字本身的分布很不均匀，比如某些范围内比较密集，则对散列函数的均匀性要求就更加突出。为了尽量减少冲突，一般要充分利用关键字的所有组成部分，即通过不同部分获得不同地址（相当于将单变量函数转化成了“多变量”函数）。

为简便起见，以下假定关键字是无符号的整型（即自然数，其他类型通常可转换为该类型），且散列地址也是无符号的整型。

1．直接定址法（Immediately Allocating Method）

将散列函数直接取为关键字的某种线性函数：

```
H(key)=a×key+b
```

若 a 为整数且不为 0，则这类散列函数是一一对应的，不会产生冲突，如例 8.1。但若关键字取值范围较大，则所需的地址范围，也即存储空间也较大，所以实际使用较少。

2．数字选择法（Digit Extraction Method）

数字选择法也称数字分析法（Digit Analysis Method）。若事先知道关键字每一位上数字的分布规律，且关键字的位数比散列地址的位数多，则可取数字分布比较均匀的若干位或其组合作为散列地址。

例如，有一组由 8 位数字组成的关键字，如图 8.21 左边一列所示。

关键字	散列地址1(0～999)	散列地址2(0～99)
99023481	248	05
99043512	451	57
99077235	723	07
99133858	385	96
99107673	067	79
99117064	106	74
…	…	…

图 8.21　关键字与散列地址示例

分析这些关键字会发现，前两位都是 99，当然不均匀；第三、五位也分别只取 0、1 和 3、7 两个值，故这些位都不可取。第四、六、七、八位数字分布较为均匀，因此，可根据散列表的长度取其中几位或它们的组合作为散列地址。比如，若表长为 1 000（即地址为 0～999），则可取其中三位（如四、六、七位）数字作为散列地址；若表长为 100（即地址为 0～99），则可取其中两位或两位的组合（如四、六与七、八位之和并舍去进位）作为散列地址等，其结果见图 8.17 中的散列地址 1 和散列地址 2。

3．平方取中法（Mid-square Method）

若关键字各位的数字分布未知，或不均匀，可采用平方取中法：先对关键字求平方以扩大差别，然后再取中间的几位或其组合作散列地址。因为乘积的中间几位数和乘数的每一位都相关，故由此产生的散列地址也比较随机（均匀），所取位数由散列表的表长决定。

例如，对关键字组{1101, 0011, 0101, 0111, 1001}，其各位分布很不均匀，平方后得：

```
{1212201,0000121,0010201,0012321,1002001}
```

若表长为 1 000，则可取中间三位作为散列地址：

```
{122,001,102,123,020}
```

4．折叠法（Folding Method）

若关键字位数较多，也可将关键字分割成位数相同的若干段（最后一段的位数可以不同），段的长度取决于散列表的地址位数，然后将各段的叠加和（舍去进位）作为散列地址。折叠法又分移位叠加（Shift Folding）和边界叠加（Boundary Folding）两种。移位叠加是将各段的最低位对齐，然后相加；边界叠加则是两个相邻的段沿边界来回折叠，然后对齐相加。移位叠加时各段的先后顺序没有作用，从这点上看不如边界叠加。

将关键字数字分段时若从低位开始，则可用取模运算取出每段的结果，如 K%100 就得到十位和个位，(K/100)%100 就得到千位和百位等。例如对关键字 62528315168，设表长为 1000，则取三位为一段进行叠加，可有如下结果：

```
     移位叠加            边界叠加
       062                 260
       528                 528
       315                 513
   +)  168             +)  168
   ---------          ---------
    [1] 073             [1] 469
  H(key)=073          H(key)=469
```

折叠法也可用于字符串，即将各字符看成 ASCII 码。特别地，如果每段只取一个字符，结果就是将各字符的 ASCII 码加起来（舍去进位）。

5．除余法（Division Remainder Method，Modulo-division Method）

选择一个适当的正整数 P，用 P 去除关键字，取所得余数作为散列地址，即：

```
H(key)=key%P
```

该方法的关键是选取适当的 P。如果 P 为关键字的基数或其幂次，则等于取关键字的低位数字作地址，与高位数字无关，显然地址不均匀，这时低位数字相同的关键字都冲突。例如，对关键字 12、512、312、1012 等，若选 P=100，则它们的地址都是 12，冲突。

如果 P 为偶数，则奇数的关键字对应到奇数地址，偶数的关键字对应到偶数地址，显然地址不均匀；特别地，若关键字集合中奇、偶数个数不等，则较多的一方更易冲突。如果 P 为奇数但不是素数，则当关键字与 P 有公因子时对应的地址也有该公因子，即地址为该公因子的倍数，也不均匀。

一般 P 取小于或等于散列表长度 m 的某个最大素数。如 m=13～16 时，可取 P=13 等。

特别地，如果散列表长度 m 本身就为素数，则取 P=m。若 m 较大，则 P 也可取合数，但一般要求其因子为较大的素数。如 P=29×31=899 等，这可避免找大素数的困难。

注意，若 P<m，则散列地址 0～P−1 后的单元不是“空闲区”，它们在以后的冲突处理中会用到（闭散列表时）。

除余法计算简单，不需要知道关键字各位的分布规律，也不必关心关键字位数的多少，并且在许多情况下效果较好，因此，除余法是一种最常用的散列函数构造方法。

6．基数转换法（Radix Transformation Method）

先把关键字看成是另一个数制上的数，再把它转换成原来数制上的数，取其中的若干位作为散列地址。一般取大于原来基数的数作为转换的基数（可扩大差别），并且两个基数要互素。例如，给定一个十进制数的关键字$(12057)_{10}$，我们先把它看成是以 13 为基数的十三进制数$(12057)_{13}$，再把它转换为十进制数：

$$(12057)_{13}=1\times13^4+2\times13^3+0\times13^2+5\times13+7=33027$$

假设散列表长度为 1 000，则可取其中三位，如低三位 027 作为散列地址。

7．随机数法（Random Number Method）

选择一个随机函数，取关键字的随机函数值作散列地址，即：

```
H(key)=random(key)
```

其中，random 为随机函数。通常，当关键字长度不等时这种方法比较恰当。

注意，随机函数应为伪随机的，即对不同关键字得到随机结果，但对同一个关键字，每次运行的结果要相同，否则建表和以后的查找过程不能保证相同的散列地址。

以上方法还可根据情况组合使用，比如除余法，若关键字连续，则散列地址也连续，此时均匀性不好，可把原关键字平方、折叠或基数转换后再用除余法。

广义地讲，各种散列函数都是在生成某个与关键字有关的“随机数”，有人用“轮盘赌”的统计分析方法进行过模拟分析，结论是平方取中法最接近于“随机化”。进一步，也可以直接借用一些随机函数的生成方法或思想来设计散列函数。

8.4.3 处理冲突的方法

处理冲突的方法基本上可以分为两大类：开放地址法和链地址法。

1．开放地址法（Open Addressing）

这种冲突处理方法的基本思想是：当发生冲突时，使用某种方法在散列表中形成一个探查序列，沿此序列逐个单元地查找，直到找到给定的关键字，或者碰到一个开放的地址（即空单元）为止[①]。插入时碰到开放地址，则将待插入的新结点存放在该地址单元中；查找时碰到开放地址，则表明表中没有待查的关键字。之所以将空单元称作**开放地址**，是因为它对所有关键字都“开放”：既可存放同义词，也可存放非同义词，取决于谁先占用它。

显然，用开放地址法建立散列表，建表前必须将表空间的所有单元置空（即设置空标志）。开放地址法组织的散列表又称为**闭散列表**，它不论记录是否冲突都存储在同一个数组

---

① 对初始散列地址的检查实际也是一次“探查”，在谈到“总探查次数”时，不仅指冲突后的探查，还应包含初始地址检查的这一次（参见习题 8.7）。

空间内，也有散列表空间是“封闭”之意。

闭散列表的表空间必须比结点的集合大，或至少相等，即装填因子 $\alpha \leqslant 1$，此时要浪费一定的空间，但换取的是查找效率。使用时，常在区间[0.65, 0.9]上取 α 的适当值。

根据形成探查序列的方法不同，解决冲突的方法也不相同。下面介绍几种常用的探查方法，并假设散列表的长度为 m，结点个数为 n，散列地址 d=H(key)。

（1）线性探查法（Linear Probing）

线性探查法的基本思想是：将散列表看成是一个环形表。若地址为 d 的单元发生冲突，则依次探查 d 的后继单元 d+1, d+2, …, m−1, 0, 1, …, d−1，即探查地址为[①]：

$d_i=(d+i)\%m \qquad 1 \leqslant i \leqslant m-1$

探查时直到找到一个空单元或关键字为 key 的单元为止。当然，若一直探查到序列的最后一个地址 d−1 都不能找到一个空单元或关键字为 key 的单元，则无论是查找还是插入都意味着失败（此时表满）。

例 8.3 已知一组关键字为{26, 15, 12, 23, 7, 64, 18, 28, 51, 29, 25}，用除余法构造散列函数，用线性探查法解决冲突，试构造这组关键字的散列表。

解：这里 n=11，尚需决定散列表长度 m 和散列函数的除数 P。在闭散列表中一般令装填因子 $\alpha<1$ 以减少冲突，现初步取 $\alpha=0.75$，则散列表长度 $m=\lceil n/\alpha \rceil=\lceil 14.6 \rceil=15$，即散列表为 HT[15]。此时实际装填因子为 $\alpha=11/15=0.733$，与预设值差别不大。用除余法构造散列函数，因 m=15，故选 P=13，即散列函数为 H(key)=key%13。

由散列函数得到各关键字的散列地址，见图 8.22（a）。

插入时，由散列地址找到相应的存储单元，若该地址是开放的，则插入新结点；否则线性探查下一个地址。第一次插入的是 26，其散列地址 d=0，是一个开放地址，故 26 插入到 HT[0]。类似，15、12、23 和 7 依次分别插入到 HT[2]、HT[12]、HT[10]和 HT[7]。

当插入 64 时，其散列地址 d=12，而 HT[12]已被关键字 12 占用，冲突，则探查下一个地址 13，为开放地址，于是将它插入 HT[13]中。接下来，18 直接插入到 HT[5]，28、51、29 和 25 均发生地址冲突，分别探查 1、2、1 和 4 次后，插入到 HT[3]、HT[14]、HT[4]和 HT[1]中。由此构造的散列表见图 8.22（b），其中的虚线表示最后插入 25 时的探查过程。

（a）

关键字 key	26	15	12	23	7	64	18	28	51	29	25
散列地址 key%13	0	2	12	10	7	12	5	2	12	3	12

（b）

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
散列表	26	25	15	28	29	18		7			23		12	64	51

（c）

查找成功时比较次数	1	5	1	2	2	1		1			1		1	2	3
查找不成功时比较次数	7	6	5	4	3	2	1	2	1	1	2	1	10		

图 8.22 线性探查法构造散列表示意图

在上例中，H(28)=2，H(29)=3，即 28 和 29 不是同义词，但由于处理 28 和同义词 15

① 一般形式为 $d_i=[d+(ai+b)]\%m$，当 i=0 时应为初始地址 d，所以 b=0；为了探查到表的所有地址，a 应和 m 互素，特别地取 a=1。

的冲突时，28 抢先占用了 HT[3]，这就使得插入 29 时，这两个本来不应该发生冲突的非同义词之间也会发生冲突。

一般地，用线性探查法解决冲突时，当表中 i, i+l, …, i+k 位置上已有结点时，散列地址为 i, i+1, …, i+k, i+k+1 的结点都将试图插入在位置 i+k+1 上。由于散列地址不同的结点，都争夺同一个后继散列地址，于是在该位置上插入的概率就比其他空位置高得多。这个过程发展下去，记录就有一种聚集到一起的倾向，称之为**堆积**（clustering）。

其结果是某些位置附近“堆积”了大量结点，造成不是同义词的结点也处在同一个探查序列之中，从而增加了探查序列的长度。若干小堆积还可能汇集成大堆积，情况会变得更差。显然，堆积越严重，以后的查找就越来越退化成顺序查找。

若散列函数选择不当，或装填因子过大，都可能使堆积的机会增加。

为了减少堆积的机会，就不能像线性探查法那样逐个探查连续的地址序列，而应该使探查序列跳跃式地散列在整个散列表中。下面的几种冲突处理方法，与线性探查法相比，可大大减少堆积的可能性。

（2）二次探查法（Quadratic Probing）

发生冲突时，探查序列依次是 $d+1^2, d+2^2$, …，即探测地址为[①]：

$d_i=(d+i^2)\%m \qquad 1\leqslant i\leqslant m-1$

通常用它的一种改进形式：探查时双向交替进行，探查序列依次是 $d+1^2$, $d-1^2$, $d+2^2$, $d-2^2$, …，即发生冲突时，将同义词来回散列在地址 d 的两端，探测地址为：

$d_{2i-1}=(d+i^2)\%m$

$d_{2i}=(d-i^2)\%m$；$if(d_{2i}<0)\ d_{2i}=d_{2i}+m$； $\qquad 1\leqslant i\leqslant (m-1)/2$

二次探查时，步伐按平方加大，可有效减少堆积的可能性，但不容易探查到整个散列表空间。只有当表长 m 为 4j+3 的素数时，才能探查到整个表空间（对改进形式而言），这里 j 为某一正整数。

可以证明，在表长为素数且装填因子不超过 0.5 时，冲突后最多再探查 1 次即可。

（3）随机探查法（Random Probing）

采用随机探查法解决冲突时，探查地址为：

$d_i=(d+R_i)\%m \qquad 1\leqslant i\leqslant m-1$

其中 R_i 为某个随机数。与构造散列函数的随机数法类似，R_i 也不能真正“随机”选取，否则建表和以后的查找过程不能保证相同的探查序列。通常取 R_i 为 1, 2, …, m−1 的一个随机排列中对应的数，并在建表和查找时按相同的排列进行探查。

二次探查和随机探查可以使非同义词的探查序列不同（但可能有交叉），但同义词的探查序列是相同的[②]。为使同义词的探查序列也不相同，探查序列就应由原关键字来决定。

---

① 一般形式为 $d_i=[d+(ai^2+bi+c)]\%m$，当 i=0 时应为初始地址 d，所以 c=0；能否探查到表的所有地址，主要取决于二次项 ai^2，简单地取 a=1，b=0（但此时并不一定能探查到所有位置）。

② 同义词因探查序列相同而引起的聚集称为**二次聚集**或次聚集(Secondary Clustering)，前述线性探查中同义词、非同义词的探查序列重叠而引起的聚集称为**一次聚集**或主聚集（Primary Clustering）。二次聚集时结点分散在表中各处，一次聚集时结点连续成片（探查步长为 1 时）。但不同文献有不同解释，如二次聚集指处理同义词冲突时出现非同义词冲突，或小聚集连成大聚集等。

（4）多散列函数和双散列函数探查法

使用多个散列函数，若前一个散列地址冲突，就使用下一个散列地址，即探查地址为：

$d_i=H_i(key)$　　　　　　　$i=1, 2, \cdots, k$

H_i为不同的散列函数。这种方法不易产生堆积现象，但计算量大。

一种改进方法是只使用两个散列函数H_1和H_2，其中H_1和前面的H一样。若$H_1(key)=d$发生冲突，则计算$H_2(key)$，用来对散列地址进行补偿。探查地址为：

$d_i=(d+iH_2(key))\%m$　　　　　$1\leqslant i\leqslant m-1$

显然H_2要永不为0（否则原地死循环探查）。定义$H_2(key)$的方法较多，但必须使$H_2(key)$的值和m互素（且小于m），才能使发生冲突的同义词地址均匀地分布在整个表中；否则，探查地址可能呈周期性变化而陷入探查死循环。

若m为素数，则$H_2(key)$取1到m−1之间的任何数均与m互素。如果H_2用除余法，则除数P不能取m−1（它为偶数，不好），一般取$H_2(key)=key\%(m-2)+1$。

若m是2的方幂，则$H_2(key)$可取1到m−1之间的任何奇数。

若m为其他情况，且$H_1(key)=key\%P$（P为m以内的最大素数），则可取$H_2(key)=key\%q+1$（q是小于P的最大素数）。

易见，线性探查和二次探查可看成特殊的双散列探查：取$H_2(key)=1$，就是线性探查；取$H_2(key)=\pm i$，则为二次探查。特别地，如果$H_2(key)$取为和m互素的常数，则相当于线性探查的一种简单改进：不是依次探查相邻的单元，而是间隔地进行。

2．拉链法（Separate Chaining）

拉链法解决冲突的做法是，将所有关键字为同义词的结点链接在同一个单链表中。若选定的散列函数的值域为0到m−1，则可将散列表定义为一个由m个头指针组成的指针数组HP[m]，凡是散列地址为i的结点，均插入到以HP[i]为头指针的单链表中。拉链法组织的散列表又称为**开散列表**，它将冲突记录存储在表外，也有散列表空间可向外扩展之意。

注意，对开散列表，装填因子$\alpha=n/m$表示的是所有同义词单链表的平均长度，显然，它可以小于1，也可以等于1或大于1。而在闭散列表中，一定是$\alpha\leqslant 1$。

对每个同义词链表，在建表时需要搜索完才知道其中是否有键值重复的记录，于是就知道了尾结点的地址，所以既可用头插法，也可用尾插法建表。对后者，表中各结点的次序就是结点的插入次序。

例8.4　已知一组关键字和选定的散列函数与例8.3相同，用拉链法解决冲突，构造这组关键字的散列表。

因为散列函数$H(key)=key\%13$的值域为0至12，故散列表为HT[13]。当把$H(key)=i$的关键字插入第i个单链表时，既可插入在链表的头上，也可以插在链表的尾上。这里每次将新关键字插入到链尾，得到的散列表如图8.23所示。

上述同义词链表还可作些改进：

（1）按关键字大小排列记录，可提高以后查找时的效率。

（2）按访问频率排列记录，可提高以后对高访问率记录的查找效率。

（3）在尾部附加一个监视哨结点，可在查找时省略结点为空的检查。

由于同义词链表一般不大，这些方法的实际意义通常并不突出。

与开放地址法相比，拉链法有如下几个优点：（1）拉链法不会产生堆积现象，因而平均查找长度较短；（2）由于拉链法中各单链表上的结点空间是动态申请的，故更适合于建

表前无法确定结点数的情况；（3）当装填因子α较大时，拉链法所用的空间会比开放地址法多，但是α越大，开放地址法所需的探查次数越大，所以，拉链法所增加的空间开销是合算的。

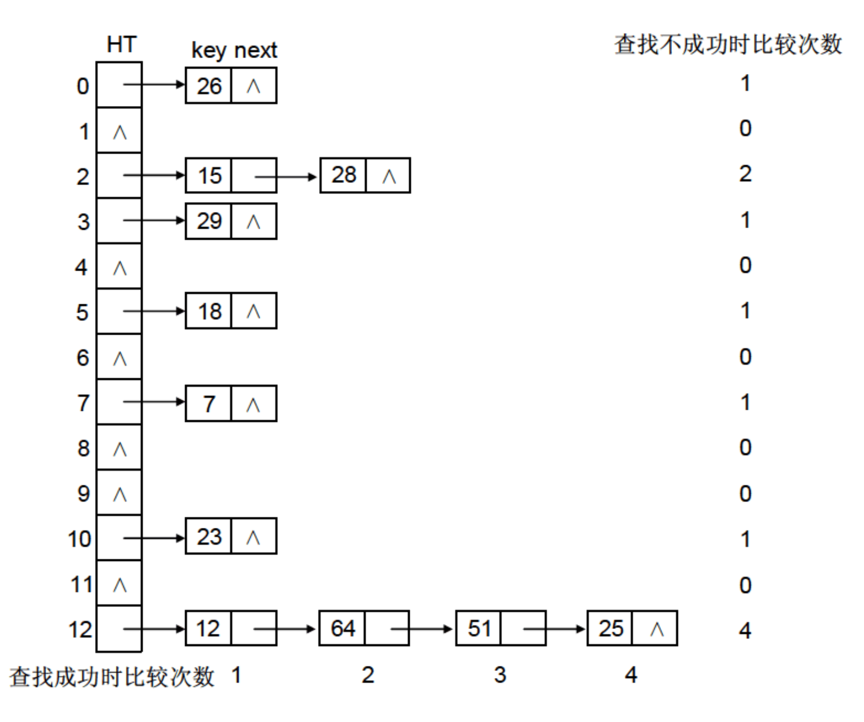


图 8.23 拉链法构造散列表示意图

8.4.4 散列表的查找及分析

散列表的运算有建表、查找、插入和删除等，但基本运算是查找。因为散列表的主要目的是为了快速查找，而建表、插入和删除均要用到查找运算。

在开散列表中，删除结点的操作易于实现，只要简单地删去链表上相应的结点即可。

但对闭散列表，删除结点时不能简单地将被删结点的空间置为空，否则将截断在它之后填入散列表的同义词的查找路径。这是因为各种开放地址法中，空单元（即开放地址）都是查找失败的条件。因此在用开放地址法处理冲突的闭散列表上执行删除操作，一般只在被删结点上做删除标记，而不真正删除结点（否则探查序列相同甚至交叉的结点可能都要前移，处理起来比较困难）。

相应地，在闭散列表中插入时，若遇到有删除标记的地址，并不能马上将当前关键字插入到该处，而应沿探查序列继续进行下去。这是因为后面的探查序列中可能已有该关键字，插入后会导致散列表中出现相同的关键字。为提高插入（以及查找）的效率，当删除标记较多时，最好对散列表重建。

散列表的查找过程和建表过程相似。以闭散列表为例，假设给定的值为 K，根据散列函数 H 计算出散列地址 H(K)，若表中该地址对应的单元为空，则查找失败；否则将该地址中结点的关键字与给定值 K 比较，若相等则查找成功，若不等则按冲突处理的方法找下一个地址，如此反复下去，直到找到某个空单元（查找失败）或者关键字相等的地址（查

找成功）为止。其中，散列函数和冲突处理的方法要和建表时设定的相同。

以下只给出散列表的查找和插入算法。

1. 闭散列表的查找和插入（不考虑删除标记）

闭散列表的类型定义及查找和插入的算法如下：

```
const keytype OPEN=0;    //空单元标志，假设为 0
const int m=20;          //散列表长，假设为 20
typedef struct {
   keytype key;
   othertype other;
} nodetype;              //结点类型
typedef nodetype hashtable[m];//闭散列表类型
int LSearch(hashtable HT,keytype K) {//在闭散列表 HT 中找关键字为 K 的结点
   int d0,d,i;
   d0=H(k);
   i=1;d=d0;
   while(i<=m && HT[d].key!=OPEN && HT[d].key!=K) {
     i++;d=(d0+i)%m;
}                        //线性探查，最多 m 次（初始地址的检查也看做 1 次探查）
   return d;             //由调用程序检测是否 HT[d]=K，若不等则查找失败
}
void LInsert(hashtable HT,nodetype s) {//在闭散列表 HT 中插入结点 s
   int d;
   d=LSearch(HT,s.key);
   if(HT[d].key==OPEN) HT[d]=s;
   else cout<<"不能插入，结点已存在或表满！\n";
}
```

2. 开散列表的查找和插入

开散列表的类型定义及查找和插入的算法如下：

```
typedef struct node * pointer; //结点指针类型
struct node{                   //结点结构
   keytype key;
   othertype other;
   pointer next;
};
typedef pointer chainhash[m];  //开散列表类型
pointer CSearch(chainhash HT,keytype K) {//在开散列表 HT 中找关键字为 K 的结点
   pointer p;
   p=HT[H(K)];                 //取出 K 所在链表的头指针
   while(p!=NULL && p->key!=K) p=p->next;//顺序查找
   return p;                   //由调用程序检测是否 p==NULL，若空则查找失败
}
void CInsert(chainhash HT,pointer s) { //在开散列表 HT 中插入结点*s
   int d;
   pointer p;
   p=CSearch(HT,s->key);
   if(p==NULL) {
     d=H(s->key);s->next=HT[d];HT[d]=s;//插入到链表头部
   }
   else cout<<"不能插入，结点已存在！\n";
}
```

3. 查找分析

从上述查找过程可知，虽然散列表在关键字和存储位置之间建立了对应关系，但是由于冲突的产生，散列表的查找过程仍然是一个和关键字比较的过程，不过散列表的平均查找长度比顺序查找要小得多，比二分查找也小。下面仍以例 8.3 和例 8.4 的散列表为例，分析在等概率情况下查找成功和不成功时的平均查找长度。

对于成功的查找，所查找的关键字只能是给定的 11 个关键字中的某一个。对图 8.22（b）所示的闭散列表，若要找关键字 26，其散列地址为 0，而 HT[0]正是它，故只需进行 1 次比较；若要找关键字 25，其散列地址为 12，而 HT[12]不是 25，向后依次探查 HT[13]、HT[14]、HT[0]和 HT[1]，最后在 HT[1]内找到，故需进行 5 次比较，类似可得到其他关键字的比较次数[①]，见图 8.22（c）的第一行。这样，线性探查法查找成功时的平均比较次数为：

ASL=(1+5+1+2+2+1+1+1+1+2+3)/11=20/11=1.82

对开散列表，见图 8.23，若要找的关键字在各个同义词链表中是第一个，则只需进行 1 次比较，如 26、7、18 等；若要找的关键字在各个同义词链表中是第二个，则需进行 2 次比较，如 28、64；类似，对关键字 51 和 25 分别要进行 3 次和 4 次比较。这样，拉链法查找成功时的平均比较次数为：

ASL=(1×7+2×2+3×1+4×1)/11=18/11=1.64

而当 n=11 时，顺序查找和二分查找的平均查找长度为：

ASL_{sq}=(11+1)/2=6

ASL_{bn}=(1×1+2×2+3×4+4×4)/11=33/11=3

对于不成功的查找，顺序查找和二分查找所需进行的关键字比较次数取决于表长，而散列查找所需进行的关键字比较次数则和待查结点有关。这里，将等概率情况下散列表在查找不成功时的平均查找长度，定义为查找不成功时对关键字需要执行的平均比较次数。

在查找不成功时，所查找的关键字可以是给定的 11 个关键字之外的任何一个，有无穷多个。为便于分析，我们将它们按散列地址分类，由于散列函数 H(key)%13 的值域为 0 到 12，故分成 13 类。对闭散列表，见图 8.22（b），假设待查关键字 K 不在该表中，若 H(K)=0，则必须依次将 HT[0]到 HT[6]中的关键字和 K 或 OPEN 进行比较之后，才发现 HT[6]为空，即比较次数为 7；若 H(K)=1，则需比较 6 次才能确定查找不成功。类似地对 H(K)=2, 3，…，12 进行分析，结果见图 8.22（c）的第二行，所以查找不成功时的平均查找长度为：

ASL_{unsucc}=(7+6+5+4+3+2+1+2+1+1+2+1+10)/13=45/13=3.46

对开散列表，见图 8.23，若待查关键字 K 的散列地址为 d=H(K)，且第 d 个链表上具有 x 个结点，则当 K 不在此表上时，就需做 x 次关键字比较（不包括空指针判定），因此，查找不成功时的平均查找长度为：

ASL_{unsucc}=(1+0+2+1+0+1+0+1+0+0+1+0+4)/13=11/13=0.846

这里的 13 既是查找不成功的种类数，又是表长。易见，该结果就等于装填因子 α。

散列表的查找效率与散列函数的均匀性、冲突处理方法和装填因子有关。如上述例子表明，散列函数相同、冲突处理方法不同的散列表，其平均查找长度是不同的。一般我们

① 这里的“比较”是指对“单元”的比较或探查，因为纯粹从关键字比较上看，对每个单元 x 比较了 2 次：是否为空以及是否为所找：x.key!=OPEN && x.key!=K。参见前述闭散列表的查找算法。

假定所选的散列函数是均匀的，这时可不考虑散列函数的影响，查找效率就只与冲突处理方法和装填因子有关。表 8.1 给出了在等概率情况下，采用 5 种不同方法处理冲突时，得到的散列表的平均查找长度。

从表 8.1 可见，散列表的平均查找长度不是结点个数 n 的函数，而是装填因子 α 的函数[①]。因此在 10 000 个结点的散列表中进行查找的平均比较次数并不一定比在 10 个结点的散列表中进行查找时大，甚至还小，只要前者的装填因子小于或等于后者即可。但其他基于比较的查找，如顺序查找、二分查找等没有这个特点，它们的平均比较次数一般随结点数的增加而增大，差别仅在于增长的快慢不同。

表 8.1 散列表的平均查找长度

解决冲突的方法	平均查找长度	
	成功的查找	不成功的查找
线性探查法	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right]$
二次探查，随机探查 双散列函数探查	$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$
拉链法	$1+\frac{\alpha}{2}$	α[②]

在具体设计散列表时可选择 α 以控制散列表的平均查找长度。例如，当 α=0.9 时，对于成功的查找，线性探查法的平均查找长度是 5.5；二次探查、随机探查及双散列函数探查法的平均查找长度都是 2.56；拉链法的平均查找长度为 1.45。显然，α 越小，产生冲突的机会就越小，查找速度就越快。但另一方面，α 越小，空间的浪费也就越多。α 值的具体选取，可在查找速度和空间耗费上进行权衡。

需要注意：

（1）表 8.1 给出的是平均情况，最坏情况显然为 O(n)，它不是装填因子 α 的函数。

（2）表 8.1 指的是大量同类问题的“平均”，对某一个具体问题，不能直接用它来计算实际结果，只能用作估计。比如，用其中的线性探查法公式计算图 8.22 的闭散列表，这时 α=11/15=0.733，则查找成功时的平均比较次数为 $\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$=2.38，而不是上面计算出的 1.82，且差别较大；又如，用其中的拉链法公式计算图 8.23 的开散列表，这时 α=11/13=0.846，则查找成功时的平均比较次数为 1+α/2=1.42，也不是上面计算出的 1.64。一般地，如果关键字个数 n 较大，且关键字分布比较均匀，则实际计算结果和公式估算结果比较接近，但对于线性探查法，当装填因子过大（如接近 1）时，两者还是可能有很大的差别，主要是因为这时有非常严重的堆积现象。

（3）对闭散列表，当 α→1 时查找长度急剧增长（趋于∞），效率极低，这是应当避免的。其中不成功查找比成功查找严重；线性探查比二次探查严重。线性探查法宜取更低的

① 严格地说，结点个数 n 有轻微影响，特别是 n 很小时（这时开放地址法在 α 较大时受 n 的影响略为明显些）。

② 若同义词链表为键值有序的，则该结果为α/2（有序表顺序查找失败时平均比较一半的结点）；若把查找失败时空指针的比较也算做一次探查，则结果要多 1 次，如 1+α等。

α（如二次探查不超过 0.9，线性探查不超过 0.8 等）。

（4）关键字出现的先后顺序对散列表平均查找长度一般影响不大或没有影响（如线性探查法和拉链法），这比二叉排序树好得多，所以散列表中一般不考虑输入顺序的影响。

显然，散列方法主要用于动态查找，但也可用于静态查找。

习 题 八

8.1 二叉树上结点的平衡因子只能为−1, 0, 1 吗？

8.2 能不能要求平衡二叉树所有结点的平衡因子都是 0？能不能要求平衡二叉树所有结点的平衡因子尽可能为 0？

8.3 n 个结点的二叉树何时高度最小？何时高度最大？平均查找长度最大是多少？

8.4 深度为 4 的 AVL 树结点数最多为多少？最少为多少？

8.5 线性表能否用散列方法存储？

8.6 何为堆积现象？它有何危害？

8.7 用线性探查法解决冲突，将 n 个同义词存入散列表时，至少要探查多少次？

8.8 对关键字序列{3, 5, 10, 12, 17, 20, 23, 27, 31, 34, 39, 40, 41}，用二分法查找关键字 12，写出查找过程，查找成功时经过了几次关键字比较？

8.9 分别对结点数 n=10 和 n=100 的有序表进行二分查找，查找成功时平均查找长度为多少？查找不成功时最大查找长度为多少？

8.10* 试求二叉排序树的平均查找长度。

8.11 （1）给定关键字集合{1, 2, 3}，试画出所有可能的二叉排序树。

（2）给定关键字集合{1, 2, …, n}，所有可能的二叉排序树有几棵？

8.12 已知图 8.24 为二叉排序树，各结点值为{16, 30, 56, 80, 20, 66}，请标出各结点的值。

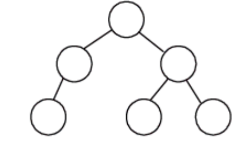


图 8.24 习题 8.12 图

8.13 已知关键字输入序列{10, 17, 8, 9, 20}，画出相应的二叉排序树，并求在等概率下查找成功和不成功时的平均查找长度。

8.14 试给出一种方法，可对任意关键字序列，构造出平均查找长度最小的二叉排序树。

8.15* 给定关键字序列{4, 5, 7, 2, 1, 3, 6}，试生成一棵平衡二叉排序树。

8.16* 已知某 3 阶 B 树如图 8.25 所示，请画出在此树上依次插入 80、32、60、46 时 B 树变化过程。

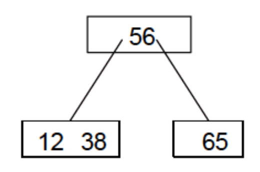


图 8.25 习题 8.16 图

8.17* 含 8 个关键字的 3 阶 B 树最多几个结点？最少几个结点？画出其形态。

8.18* 对关键字序列{23, 15, 7, 47, 28, 14, 26, 53, 32, 69, 81, 77}构造散列表，设散列表空间为 HT[0..14]，用除余法构造散列函数，用二次探查法解决冲突，画出相应的散列表，并求查找成功时的平均查找长度。

8.19* 设散列表有 200 个表项，用二次探查法解决冲突，查找不成功时插入新表项的平均探查次数不超过 1.5，试设计一个合适的散列表长度和相应的除余法散列函数。

8.20　对关键字序列{11, 78, 10, 1, 3, 2, 4, 21}构造散列表，取散列地址为 HT[0..10]，散列函数为 H(K)=K%11，试分别用线性探查法和拉链法解决冲突，画出相应的散列表，并分别求查找成功和不成功时的平均查找长度。若直接用公式计算，结果如何？

8.21　假设对顺序表进行从前向后的顺序查找，试写出相应的算法。

8.22　试写出链表上的顺序查找算法。

8.23　试写出二分查找法的递归算法。

8.24　怎样使二叉排序树的结点按从大到小的顺序输出？

8.25　编写算法，判断二叉树是否为二叉排序树。

8.26　试编写非递归算法，分别对二叉排序树：（1）插入；（2）查找。

8.27　某闭散列表装填因子小于 1，散列函数为关键字第一个字母在字母表中的序号，按线性探查法解决冲突。试编写算法，按第一个字母的顺序输出散列表的所有关键字。

8.28　试编写算法，求等概率下开散列表中查找不成功时的平均查找长度。

第9章 文件

前面各章讨论的数据结构都是针对内存（内部存储器）的。但是，如果数据的“规模”很大（数据量大，包含的数据元素多）、或者需要长期保存，则必须存放在外存（外部存储器）中。通常将存放在外存中的数据称为文件，文件与外存密切相关。外存与内存储器的物理特性不同，其使用也有很大不同，前面介绍的各种数据组织方法和操作方法不能简单地照搬过来，需要根据文件的特点专门加以考虑。

本章讨论文件的逻辑特性、存储结构等问题，包括顺序文件、索引文件、索引顺序文件、散列文件和多关键字文件。

9.1 文件的基本概念

9.1.1 文件结构

文件（File）是外存中性质相同的若干记录的集合，每个记录由一个或多个数据项构成。记录也就是以前所说的数据元素，它是文件存取的基本单位，数据项是文件可使用的最小单位。数据项有时也称为**字段**、**域**（Field）或**属性**（Attribute）。可见，数据结构中所讨论的文件是指数据库意义上的文件，记录是有结构的，而不是操作系统意义上的文件。操作系统中研究的文件是一维无结构的连续字符序列，提供对文件的整体操作（如打开文件、关闭文件、删除文件、复制文件等）和字节操作（从文件读一字节、写一字节到文件中等）。另外，数据结构讨论的文件是指文件结构，而不是数据库本身。

例如，图 9.1 是一个简单的职工档案文件，每个职工的信息组成一个记录，单位有多少人，文件就有多少条记录。每个记录由 4 个数据项组成：职工号、姓名、性别、年龄等，每个数据项表示职工的某方面的信息，其中“职工号”可作为主关键字，而姓名、性别等只能作为次关键字。

职工号	姓名	性别	年龄
0001	王　刚	男	30
0002	田晓峰	男	25
0005	李　芳	女	27
0006	张建军	男	24
⋮	⋮	⋮	⋮

图 9.1　文件示例

这里主/次关键字的概念，见排序一章所述。

对用户来说，文件记录应是存取的基本单位。但外存设备（如磁盘、磁带）一般是按一定大小的物理块（常称作页块）存取的，它是外存存取的基本单位，一般可存放多个记录。为了区分这两种标准，通常将按用户观点看的基本存取单位（即记录）称为**逻辑记录**，将按外存设备观点看的基本存取单位称为**物理记录**。

文件可以按照记录中关键字的多少，分成单关键字文件和多关键字文件。若文件中的记录只有一个关键字（为主关键字），则称**单关键字文件**；若文件中的记录除了主关键字外，还含有若干个次关键字，则称为**多关键字文件**。

文件还可分成定长文件和不定长文件。若文件中各记录含有的信息长度相同，则这类记录称为定长记录，由定长记录组成的文件称为**定长文件**；若文件中各记录含有的信息长度不等，则称为**不定长文件**。图 9.1 所示的职工文件是一个定长文件。

类似其他数据结构，文件结构也包括逻辑结构、存储结构及文件上的各种操作（运算）这三个方面。其中，文件操作定义在逻辑结构上，操作的具体实现在存储结构上进行。

1．文件的逻辑结构及操作

文件是记录的集合，即其逻辑结构为集合。但一个文件的各个记录一般会按照某种次序排列起来（这种排列的次序可以是记录中关键字的大小，也可以是各个记录存入该文件的时间先后等），各记录之间就自然地形成了一种线性关系。因此，文件也可看成一种线性结构，或者说是存储在外存上的线性表。

类似于线性表，文件的基本运算有：读、写、定位、插入、删除等，它们可归为两类：**检索**和**修改**。除了基本运算，文件操作还有：为提高文件效率进行的再组织、文件被破坏后的恢复、文件中数据的安全保护等，这些内容和文件的修改可统称为维护，所以，从大的方面看，也可把文件的操作分为如下两类：**检索**和**维护**。

检索就是在文件中查找满足给定条件的记录，它既可以按记录的逻辑号（记录进入文件时的顺序编号）查找，也可按关键字查找。这种在外存中读取或定位记录的方式称为存取方式。基本的存取方式有以下几种：

（1）顺序存取。依次存取一个逻辑记录。在 k 号记录读取之前，序号比 k 小的记录必须已被读取过。

（2）直接存取（随机存取）。直接存取第 i 个逻辑记录，不需等待其前面记录的读取。

（3）按关键字存取。存取键字与给定值相等（或相关）的记录。对数据库文件可以有如下 4 种查询方式：

① 精确查询。查询关键字等于给定值的记录。例如，查询已知职工号或姓名的记录。

② 范围查询。查询关键字在某个范围内的记录。例如，查询 25 岁到 30 岁的职工。

③ 函数查询。给定关键字的某个函数进行查询。例如，查询所有职工的平均年龄。

④ 组合查询。查询关键字满足多个条件的记录。例如，查询 30 岁以上的男职工。

修改是指对文件进行记录的插入、删除及更新等操作。插入、删除是针对整条记录而言的，如对图 9.1 所示文件，若王刚调到了其他单位，则在文件中把他的记录删除；赵亮新分配到了本单位，则在文件中增加他的记录。记录的更新是针对记录的字段而言的，如发现李芳的年龄弄错了，不是 27 应该为 25，将它改正过来等。

文件的检索和修改都有实时和批量两种处理方式。实时处理的时间要求较严，一般要

在短时间内（如几秒钟内）迅速完成；批量处理的时间要求较松，可以在定期或不定期的一段时间后进行。例如，民航、铁路售票系统，检索和修改都应当实时处理；而银行的账户系统需要实时检索，但可进行批量修改，即可以将一天的存款和提款信息记录在一个事务文件上，在一天的营业之后再进行集中处理。

2．文件的存储结构

文件的存储结构（亦称物理结构），是指文件在外存上的组织方式。文件的组织方式很多，采用不同的组织方式就得到不同的存储结构。基本的组织方式有 4 种：顺序组织、索引组织、散列组织和链组织，其他方式往往是这 4 种基本方式的结合。按这 4 种方式组织的文件分别称为顺序文件、索引文件、散列文件和链式文件。其中链式文件常常结合索引文件一起使用，如多关键字文件。链式文件的链结点一般很大且不定长，其链结点除包括结点本身的内容和链指针（地址）外，还包括结点长度（对不定长结点）。

选择哪一种文件组织方式，取决于对文件中记录的使用方式和使用频繁程度、存取要求、外存的性质和容量等。

评价一个文件组织的效率，是执行一个文件操作所花费的时间和文件组织所需的存储空间。通常文件组织的主要目的，是为了能高效、方便地对文件进行操作，而检索功能的多少和速度的快慢，是衡量文件操作质量的重要标志，因此，如何提高检索的效率，是研究各种文件组织方式首先要关注的问题。

9.1.2 外存储器简介

外存储器简称外存，是内存的下级存储器，用于存储大量数据和永久保存，还可用作数据缓冲。计算机中 CPU 的操作，一般针对的是内存中的数据。涉及外存的操作，一般只是成块的输入和输出（传输）。为了讨论文件的组织方式，有必要介绍一下外存储器特别是磁带和磁盘的一些知识。

外存设备大体上可分为顺序存取（如磁带）和直接存取（如磁盘）两大类。现在常见的外存主要有磁盘（软盘、硬盘）、磁带、光盘、闪盘（俗称 U 盘）等。其中硬盘很长时间以来都是主流外存，在操作系统或应用系统中，它是内存的直接延伸，应用程序和操作系统的活动数据，都存储在硬盘中。其他外存一般作后备存储器。对于海量（小量）数据的储存和备份，以前常用磁带（软盘），现在则被光盘（闪盘）逐渐取代。磁盘、磁带利用磁介质的磁化与否记录信息，光盘利用激光对特殊材料“刻痕”、 闪盘对闪存进行电擦写（无需机械式装置），后两者超出本书范围，这里不做介绍。

1．磁带

磁带存储器用磁带记录信息，磁带机可以控制磁带前进、后退，磁带机上的读写磁头可以读写磁带上的信息。磁带的运行情况类似于录音机磁带的运行，如图 9.2 所示。

磁带有不同的规格，如带宽（1/4 英寸、1/2 英寸等）、带长（2 400 英尺、1 200 英尺等）、存储密度（1 600 位/英寸、3 200 位/英寸等）、磁道数（9 道、16 道等）等。常用的磁带一般为 1/2 英寸宽，最长可达 3 600 英尺。图 9.3 是一段 9 道磁带，带面在横向每排可记录 9 位二进制信息，其中 8 位组织成一个字节，另一位为奇偶校验位。

磁带上的信息是以块为单位存放的。一个信息块由若干字节构成，如 1KB～8KB。一

个信息块就是磁带存储器的一个物理记录。通常一个信息块可存放多个逻辑记录。要读写某一个块上的信息，首先要定位，即通过磁带的卷动使磁头对准被读块的前端。磁带不是连续运转的设备，而是一种启停设备。为适应启动时的加速和停止时的滑动，磁带上块与块之间需要留出**间隙**（Inter Block Gap）。间隙通常 1/4～3/4 英寸长。间隙是一段空白区，不存放数据信息。

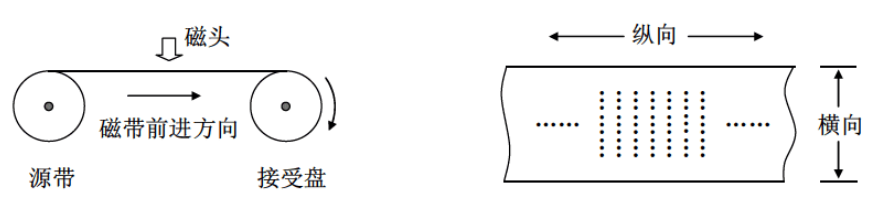


图 9.2　磁带的运行示意图　　图 9.3　9 道带示意图

磁带存储器是一种顺序存储设备，它的主要缺点是读写速度慢。磁带存取速度取决于磁带的存储密度和走带速度，实际上磁带花在定位上的时间往往比较长，如果磁头离所找的块很远时，往往要几分钟甚至十几分钟才能定位。因此磁带存储器适合于顺序存取，即读写一块之后，下一次读写它后边的相邻块，这样可以减少定位时间。

2．软盘

软盘是通过在圆形软质塑性材料上涂上磁性物质形成的（形成磁介质）。磁性物质可通过电的作用处于“磁化”和“未磁化”两种状态。两种状态分别用来表示二进制 1 和 0。软盘一般有 3 英寸和 5 英寸、单面和双面、低密和高密之分，如以前常见的一种双面高密 3 英寸软盘，容量为 1.44MB。

为了方便使用磁介质，将盘面逻辑划分为若干同心圆（磁盘有点像唱片，但唱片是螺旋线），每个同心圆称为一个**磁道**。圆形盘面上再逻辑划分为若干**扇区**。这样，每个磁道被扇区划分为若干扇段，如图 9.4 所示。一个扇段一般可存储若干字节。扇段一般是 CPU 或其他设备读写的基本单位（存储单元）。

扇段

图 9.4　盘面示意图

磁道和扇段都编号使用。按磁道和扇段的观点，每个盘面是一个二维存储区。每个存储单元（扇段）的地址为（磁道号，扇区号）。

在实际使用中，也经常依某种方式，将此二维结构映射为一维。比如，可以给 0 道 0 扇区的扇段编号为 0，0 道 1 扇区的扇段编号为 1，……，其余类推，0 道编完后，接着依次为其他道（1 道、2 道、……）上的扇段编以连续的号。这样，二维结构就映射为一维结构了。

3．硬盘

在硬盘中有若干个盘片，它们通过一个主轴串在一起，构成一个盘组，其中第一片和最后一片向外侧的一面一般不存放信息。每个盘片采用刚性材料以适应高速旋转的要求。各个盘面上半径相同的磁道合在一起称作一个**柱面**，不同半径的磁道的数目，就是柱面的数目。盘组有多少个盘面，则说柱面有多少个磁道。柱面、盘面、磁道都编号使用。其中，磁道有两个编号：为了指明它在所在盘面上的位置，需要一个盘面上的编号；为了指明它在所在柱面上的位置，又需要一个柱面上的编号。前者就是柱面号，后者就是盘面号。

因此，盘组上的存储单元（扇段）的地址为三维结构（柱面号，盘面号，扇区号）。

每个盘面都设置读写磁头来读写各自的数据。读写磁头有两种类型，一种是固定头，即每个盘的每个磁道都对应着一个专用的磁头，这种方式特点是速度快，但结构复杂，造价高。目前广泛使用的是活动头，即每个盘面只对应一个磁头，安放在活动臂上，通过活动臂的进退找到指定柱面上的磁道。所有磁头在每一时刻总是对准同一个柱面上的各个磁道。图 9.5 是磁盘组的示意图。

读写盘组上的信息，首先要经过定位动作：

（1）选定柱面。通过磁头臂移动使磁头对准指定的柱面。这是机械动作，包括磁头臂的启动、移动、停止，目前平均要几毫秒至十几毫秒，速度慢。

（2）选定磁道（即盘面）。即选择对应着所需盘面的磁头，这由电子线路实现，速度快。

（3）选定扇段（即物理记录）。磁头定位到要读写的扇段，这是机械动作，通过盘组的旋转实现，磁头只是等待需要的扇段旋转到自己下面。因为盘组的旋转不需要停止，故比寻道时间少，但一般也是毫秒级。

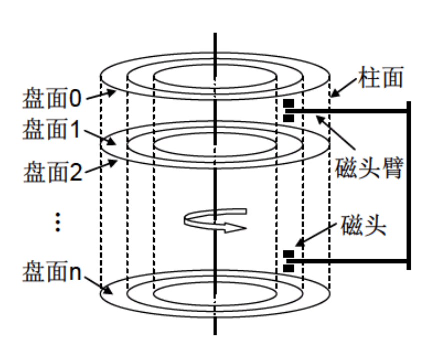


图 9.5　磁盘组示意图

经过定位后，真正用于读写信息的时间比定位时间少得多。

与磁带存储器相比，磁盘（特别是硬盘）存储器的优点是存取速度快，既适应于顺序存取，又适应于随机存取。

4．缓冲技术

由于外存速度慢，为了提高数据读写速度，可尽量减少访问外存次数。这可采用缓存（Caching）或缓冲（Buffering）技术：在每次访问外存时，顺便将更多的数据读入内存的一个暂存区域（称为缓冲区），这样下次访问其他数据时，有可能在已读入的数据中进行，而不必再次读外存。缓冲或缓存技术是外设（包括外存）使用中的一项关键技术，它使主机不必将大量的时间浪费在等待慢速的外设上。

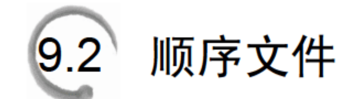
外设（包括磁盘等外存）往往可与 CPU 并行工作，因此，可在 CPU 使用缓冲区的同时，也让外设对缓冲区读写数据，以进一步提高访问速度。如果存在共享资源（缓冲区）的访问冲突，则可设立多个缓冲区。这多个缓冲区称为缓冲池。

采用缓冲技术后，主机对外存数据并不进行直接的存取，如要读外存上的数据，首先由有关通道将数据读到缓冲区，然后从缓冲区读取数据；写数据时，将数据送到缓冲区，再由有关通道将数据写到外存。一次从外存读数据或向外存写数据的过程称作一次访外。一次访外可传送若干字节，访外时间包括定位和传送时间，节省存取时间的一个有效方法是使每次访外时在内存和外存之间传送较大一批数据，从而减少访外次数。

分页块的存储方法是一种有利于减少访外次数又便于管理的方法，一个页块是磁带或磁盘上的一个物理记录，它包括多个逻辑记录，内存中设置的缓冲区应该和页块的大小相等。每次访外是把一个页块读入缓冲区或把缓冲区写入页块。若一次访外所传送的页块上有多个要在近期进行处理的逻辑记录，则分页块的存储方法可以使访外次数大大减少。

这里我们可用**访外次数**作为衡量检索效率的一个主要参数。检索一次，访外次数越少，效率就越高；反之，效率就越低。另一个衡量检索效率的参数是**磁头定位时间**。检索某一

记录，磁头定位时间越少，效率就越高；反之，效率就越低。



9.2 顺序文件

顺序文件（Sequential File）是文件的一种常见组织形式。在顺序文件中，记录按进入文件（外存）的先后顺序存放，其逻辑顺序和物理顺序一致。若顺序文件中的记录按其主关键字有序，则称此顺序文件为**顺序有序文件**；否则称为**顺序无序文件**。为了提高检索效率，常常将顺序文件组织成有序文件，本节也假定顺序文件是有序的。

顺序文件要求占用一片连续的外存区域。这里的连续是指外存访问意义下的连续，即物理记录的连续。一般情况下，外存可视为扇段的连续体。在特殊情况下，也按柱面、盘面、扇段的读写顺序安排记录。同一个柱面上的同一盘面号上的记录，认为是连续的。如果文件较大而外存空间又比较零乱，可能需要先借助有关工具进行碎片整理。一般操作系统会尽量将属于同一文件的内容安排在一片连续区中。

顺序文件适合于顺序存取和成批处理。顺序文件特别适合磁带存储器，也适应磁盘存储器。顺序文件的检索操作方法如下：

（1）顺序存取。从文件的第一个记录开始依次逐个地读入各个记录进行处理和使用。要检索第 i 个记录，必须检索它之前的 i–1 个记录。这对少量的检索是不经济的，但适合于批量检索（类似下面的批量修改），这时效率很高，因为省去了许多磁头定位的时间。

（2）随机存取。对顺序文件的这种检索方式，由于没有建立逻辑记录和物理记录之间的直接对应关系，每次检索都要从文件的第一记录开始找到所要处理的记录。这样要花费许多定位时间，因此顺序文件的随机存取效率很低。

（3）按关键字存取。即查找指定键值的记录。最简单的方法是顺序查找，从文件头到文件尾扫描每个记录，若找到则停止；否则，继续查找，直到文件尾。若检索各个记录的概率相同，则一次检索平均要扫描文件一半的记录。设文件占用的页块数为 N，则平均访外次数为 N/2，速度很慢。

一切存储在顺序存取存储器（如磁带）上的文件只能是顺序文件，且只能进行顺序查找。存储在直接存取存储器（如磁盘）上的文件可以是顺序文件，也可以是其他文件（如索引文件）。对于磁盘顺序文件，除了顺序查找外，若是有序的，还可以用分块查找、二分查找、插值查找等。其中，二分查找只能对较小的文件或一个文件的索引进行查找，当文件很大，在磁盘上占有多个柱面时，二分查找将引起磁头来回移动，增加寻道时间。

顺序文件的修改操作比较困难，它不能像顺序表那样进行插入、删除和更新（若更新关键字，则相当于删除后再插入），因为文件中的记录不能像向量空间的数据那样“移动”，只能通过复制整个文件的方法来实现。为了减少操作的代价，通常采用批量处理的方式来完成。这一方式需要引入一个附加文件（常称为**事务文件**），用来存放对顺序文件（又称主文件）的修改请求。当修改请求积累到一定数量、事务文件变得足够大时，开始实施批量处理：首先将事务文件按主关键字排序，再根据事务文件对主文件进行一次全面的修改，产生一个新的主文件；然后，清空事务文件，以便用来积累此后的修改请求。上述修改过程类似于对事务文件和主文件执行二路归并（注意事务文件和主文件都是有序的），基本步

骤是：同时对事务文件和主文件扫描。扫描中，对事务文件中的每一个请求 Q：若 Q 是删除或更新请求，则当扫描到主文件中与 Q 键值相等的记录 R 时依 Q 对 R 进行删除或更新；若 Q 是插入请求，则等主文件扫描到适当位置时执行插入动作。

顺序文件的主要优点是连续存取的速度较快，适宜于顺序存取和成批处理。若顺序文件中第 i 个记录刚被存取过，而下一个要存取的是第 i+1 个记录，则这次存取将会很快完成。若顺序文件存放在单一存储设备（如磁带）上时，这个优点总是可以保持的，但若它是存放在多路存储设备（如磁盘）上时，则在多道程序的情况下，由于别的用户可能使磁头移向其他柱面，就会降低这一优点。因此，顺序文件多用于磁带。

9.3 索引文件

用索引的方法组织文件时，通常是在文件本身（称为主文件）之外，另外建立一张表，它指明逻辑记录和物理记录之间的一一对应关系，这张表就叫做索引表。由索引表和主文件两部分一起构成的文件称为**索引文件**，它在存储器上分为两个区：索引区和数据区。使用索引的目的是为了快速访问记录，由于文件数据量大，且存储在慢速设备上，这一目的更加突出，特别是在数据库系统中，更是广泛使用了索引。

索引表中的每一项称作索引项，索引项一般是由关键字（或逻辑记录号）与相应记录的物理地址组成的。显然，索引表必须按关键字有序，若主文件本身也按同一关键字有序，则称为**索引顺序文件**（Indexed Sequential File），否则称为**索引非顺序文件**（Indexed Nonsequential File）。如果没有特别强调，通常所说的索引文件一般指索引非顺序文件，本节只讨论这种文件。

对于索引非顺序文件，由于主文件中记录是无序的，必须为每个记录建立一个索引项，这样建立的索引表称为稠密索引。对于索引顺序文件，由于主文件中记录按关键字有序，则可对一组记录（如对应一个物理记录）建立一个索引项，这种索引表称为稀疏索引。

在建立文件数据的同时，系统按用户要求自动建立索引表。开始时，索引项按记录的先后顺序排列，此时索引表是无序的，待全部记录输入完毕后，再对索引表进行排序。例如，对于图 9.6（a）的数据文件，主关键字是职工号，排序前的索引表如图 9.6（c）所示，排序后的索引表见图 9.6（b），图 9.6（a）和图 9.6（b）一起组成索引文件。

物理地址	职工号	姓名	性别
1000	04	张　娟	女
1010	11	王志刚	男
1020	06	李　明	男
1030	18	周　晶	女
1040	15	汪　平	男
1050	09	赵雪芹	女
1060	20	蔡　林	男

（a）文件数据区

关键字	物理地址
04	1000
06	1020
09	1050
11	1010
15	1040
18	1030
20	1060

（b）索引表

关键字	物理地址
04	1000
11	1010
06	1020
18	1030
15	1040
09	1050
20	1060

（c）输入过程中建立的索引表

图 9.6　索引非顺序文件示例

索引文件的检索方式为直接存取或按关键字存取。整个过程分两步进行：首先查找索

引表，若该记录在表上存在，则根据索引项指示的物理位置到外存上读取；否则该记录不存在。在找索引表时，要将外存上含有索引区的页块读入内存。在外存上读取记录时，又要将含有记录的页块读入内存。若索引表不大，则可将索引表一次读入内存，因此，索引文件的检索一般只需两次访问外存：一次读索引，一次读记录。由于索引表是有序的，对索引表的查找可用顺序查找，也可用二分查找等方法。

注意，如果只需知道某个记录是否存在以及在何处，并不要真正将它读出来（这种情况称作**预查找**），则在稠密索引表中就可完成，此时只需一次访外。稀疏索引表不能进行预查找，但它占用的空间少。

索引文件的修改比较容易实现。删除一个记录仅需删去相应的索引项；插入一个记录时，将记录置于数据区的末尾，同时在索引表中插入索引项；更新记录时，应将更新后的记录置于数据区的末尾，同时修改索引表中相应的索引项。

当文件中记录数目很大时，索引表也很大，以致一个物理页块容纳不下。这时查阅索引仍要多次访问外存。为此，可以对索引表再建立一个索引，称为**查找表**。因为索引表是有序的，查找表可进行分块索引（稀疏索引），如对每个物理块建一个索引。这时索引文件的检索要 3 次访外：查找表→索引表→数据文件。如果查找表还很大，可再对其建索引，依次类推，即进行多级索引。通常最高可达四级索引：第三查找表→第二查找表→查找表→索引表→数据文件。检索过程从最高一级索引即第三查找表开始，需要 5 次访外。

上述多级索引是一种静态索引，各级索引均为顺序表，结构简单，但修改很不方便，每次修改都要重组索引。因此，当数据文件在使用过程中记录变动较多时，应采用动态索引，例如二叉排序树（或 AVL 树）、B 树（或其变型），这些都是树表结构，插入、删除都很方便。又由于它们本身是层次结构，因而实际上无须建立多级索引，而且建立索引表（树表）的过程即相当于排序过程。通常，当数据文件的记录数不很多，内存容量足以容纳整个索引表时，可采用二叉排序树（或 AVL 树）作索引；当文件很大时，索引表（树表）本身也在外存，则查找索引时需多次访问外存，并且访问外存的次数恰好为查找路径上的结点数。显然，为减少访问外存的次数，就应尽量缩减索引表的深度。此时可采用 m 叉的 B 树（或其变型）作索引表，m 的选择取决于索引项的多少和缓冲区的大小。

总之，因为访问外存的时间比内存中查找的时间大得多，所以，评价外存中索引表的查找性能，主要着眼于访问外存的次数，即索引表的深度。注意，索引文件只能是磁盘文件，因为索引文件的组织方式是为随机存取而设计的，而磁带的随机存取效率很低。

9.4 索引顺序文件

上节介绍的索引非顺序文件适合于随机存取，不适合顺序存取。因为主文件是无序的，顺序存取将引起磁头的频繁移动。但索引顺序文件的主文件是有序的，它既适合于随机存取，也适合于顺序存取。另外，索引非顺序文件的索引是稠密索引，而索引顺序文件的索引是稀疏索引，后者占用的空间较少。因此，常用的文件组织形式是索引顺序文件。本节介绍两种最常用的索引顺序文件：ISAM 文件和 VSAM 文件。

9.4.1　ISAM 文件

ISAM（Indexed Sequential Access Method：**索引顺序存取方法**）是一种专为磁盘存取文件设计的文件组织方式，采用静态索引结构。在采用 B 树之前，IBM 曾经广泛地使用它。由于磁盘是以盘组、柱面和磁道三级地址存取的设备，则可对磁盘上的数据文件建立盘组、柱面和磁道多级索引，下面只讨论在同一个盘组上建立的 ISAM 文件。

ISAM 文件由多级主索引、柱面索引、磁道索引和主文件组成。为了提高访问效率，文件的记录在同一盘组上存放时，应尽量先集中放在同一个柱面上，然后再顺序存放在相邻的柱面上。对同一柱面，则应按盘面的次序顺序存放。例如图 9.7 所示为存放在同一个磁盘上的一个 ISAM 文件，其中 C 表示柱面，T 表示磁道（图中每个磁道存放 4 条记录），C_iT_j 表示 i 号柱面 j 号磁道，R_i 表示关键字为 i 的记录。

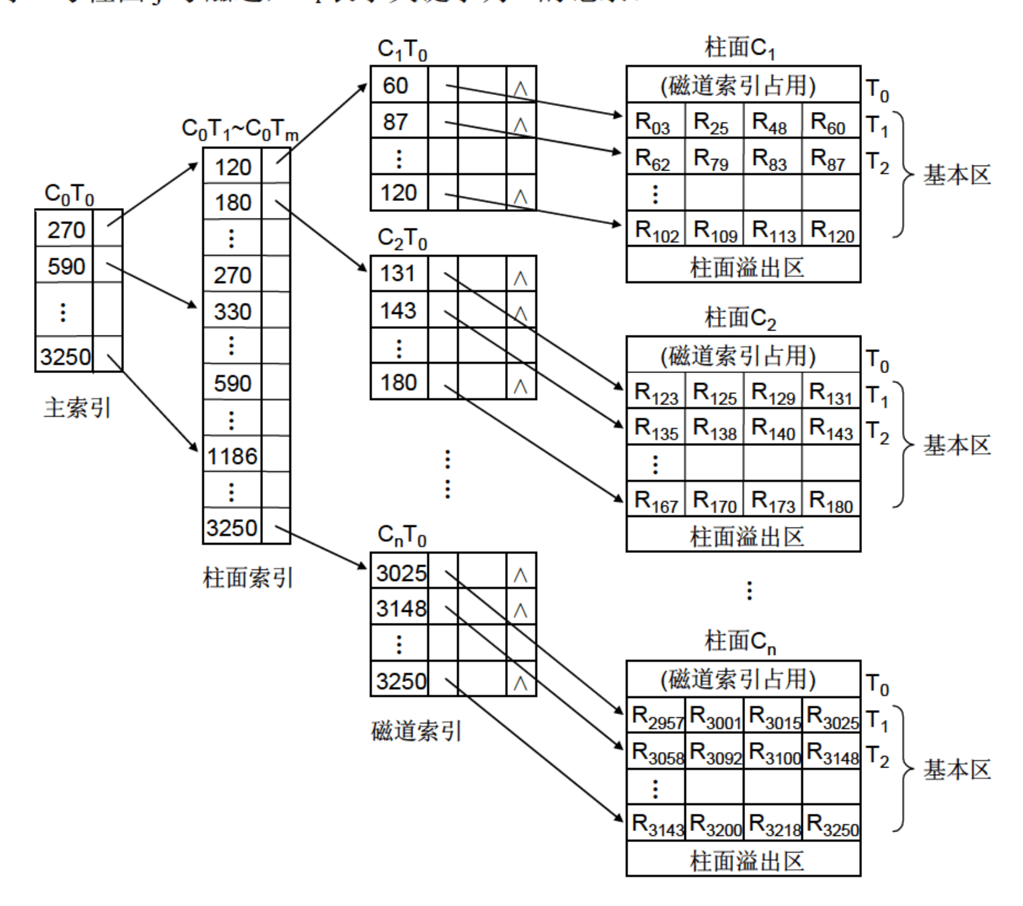


图 9.7　ISAM 文件结构示例

图 9.7 的文件只有一级主索引，它是柱面索引的索引。如果柱面索引很大，使得主索引也很大时，可对主索引再建索引，组成多级主索引。当然，如果柱面索引较小，主索引也可省略。通常主索引和柱面索引放在同一个柱面上（图 9.7 中放在 0 号柱面，但为了减少磁头在柱面间的来回移动量，放在文件中间位置的柱面较好），主索引放在该柱面最前的一个磁道 T_0 上，其后的磁道中存放柱面索引。

每个存放主文件的柱面都建立一个磁道索引，放在该柱面最前面的磁道 T_0 上，其后的

若干个磁道是存放主文件+的基本区，该柱面最后的若干个磁道是柱面溢出区（若柱面溢出区不足，还可设置一个公共溢出区）。基本区中的记录按关键字的大小顺序存储，溢出区被该柱面上基本区中的各磁道共享，当基本区中某磁道溢出时，就将其溢出记录，按主关键字大小链成一个链表（以下简称溢出链表），放入溢出区。

各级索引中的索引项结构，如图 9.8 所示，其中，每个磁道索引的索引项由两个部分组成：基本索引项和溢出索引项。

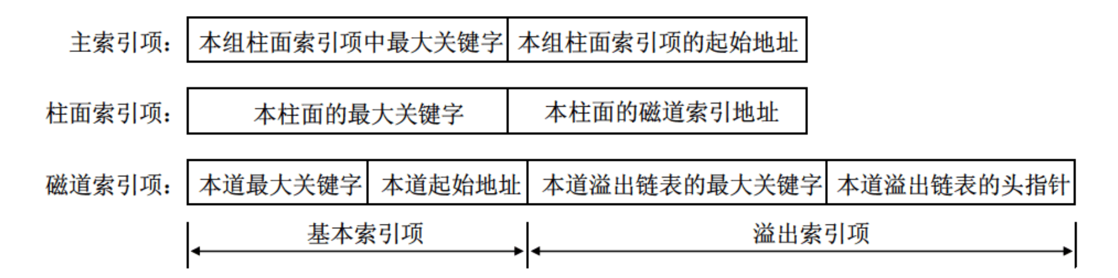


图 9.8　各级索引项格式

在 ISAM 文件上检索记录时，先从主索引出发，找到相应的柱面索引，再从柱面索引找到记录所在柱面的磁道索引，通过其基本索引项或溢出索引项，找到记录所在磁道的起始地址或溢出链表的头指针，由此出发，在该磁道上或溢出区中进行顺序查找，直到找到或找遍为止。若找遍该磁道或溢出区都没有找到此记录，则文件中无此记录。

例如，要在图 9.7 中查找记录 R_{129}，先查主索引，即读入 C_0T_0，因为 129<270，则查找柱面索引的 C_0T_1，即读入 C_0T_1，因为 120<129<180，所以，进一步把 C_2T_0 读入内存，查磁道索引，因为 129<131，所以，C_2T_1 即为 R_{129} 所存放的磁道，读入 C_2T_1 后即可查得 R_{129}。若找 R_{130}，过程类似，但最后找遍该磁道都没有，则文件中无 R_{130}。

为了提高检索效率，通常可让主索引常驻内存，并将柱面索引放在数据文件所占空间居中位置的柱面上，这样，从柱面索引查找磁道索引时，磁头移动距离的平均值最小。

当插入新记录时，由于记录要按关键字顺序存放，这时需移动记录。首先找到应插入的磁道，若该磁道已满，则磁道上最末一个记录移至溢出区，同时修改磁道索引的基本索引项和溢出索引项。例如，将记录 R_{36} 插入到图 9.7 的文件，它应插在 C_1 柱面 T_1 磁道上，结果该磁道上最后一个记录 R_{60} 被移入溢出区。此时该磁道上最大关键字由 60 变成了 48，同时溢出链表也由空变为含有一个记录 R_{60} 的表。因此，将 C_1T_0 对应的磁道索引项中基本索引项的最大关键字，由 60 改为 48；将溢出索引项的最大关键字置为 60，且令溢出链表头指针指向 R_{60} 的位置。类似地，如果再插入 R_{68} 和 R_{77}，则 C_1 柱面 T_2 磁道上的 R_{87} 和 R_{83} 被先后移到溢出区，即该磁道的溢出链表上有两个记录。于是也要分别修改磁道索引中相应的基本索引和溢出索引，最后结果见图 9.9。注意溢出索引指针指向 R_{83}，而 R_{83} 又指向 R_{87}，这是为了保证溢出链表按关键字有序。

ISAM 文件中删除记录的操作，比插入简单得多，只要找到待删除的记录，在其存储位置上作删除标记即可，而不需要移动记录或改变指针。在经过多次的增删后，文件的结构可能变得很不合理。此时，大量的记录进入溢出区，而基本区中又浪费很多空间。因此，通常需要定期地整理 ISAM 文件，把记录读入内存，重新排列，再复制成一个新的 ISAM 文件，填满基本区而空出溢出区。

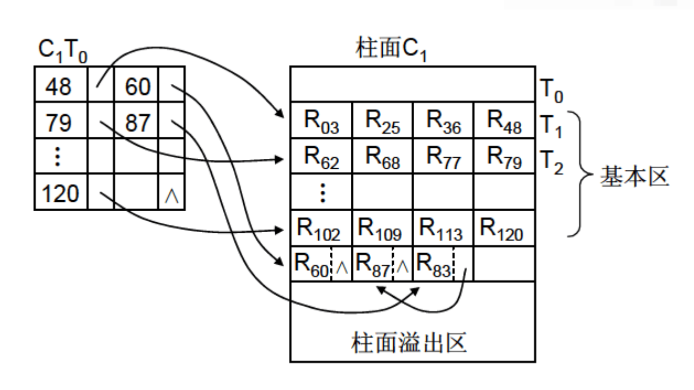


图 9.9　插入 R_{36}、R_{68}、R_{77} 后的状况

9.4.2　VSAM 文件

VSAM（Virtual Storage Access Method，**虚拟存储存取方法**），也是一种索引顺序文件的组织方式，采用 B^+树作为动态索引结构。之所以称其为“虚拟存取”，是指对用户来说，存储单位是“逻辑”的（即下述控制区间和控制区域），与存储设备无关（与柱面、磁道等物理存储单位没有必然的联系）。例如，可以在一个磁道中放 n 个控制区间，也可以一个控制区间跨 n 个磁道；在具体存取某个记录时，不需要考虑该记录当前是否在内存，也不需要考虑何时访外。IBM 公司 VSAM 文件是用 B^+树作为文件的稀疏索引的一个典型例子，它使用了 IBM 370 系列的操作系统的分页功能，存取方法与存储设备无关。

如果 B^+树中每个叶结点中的关键字对应一个记录，则适宜于作稠密索引。若让叶结点中的关键字对应一个页块，则 B^+树可用来作为稀疏索引。VSAM 文件的结构如图 9.10 所示。它由三部分组成：索引集、顺序集和数据集。文件的记录均存放在数据集中，数据集中的一个结点称为**控制区间**（Control Interval），它是一个 I/O 操作的基本单位，每个控制区间含有一个或多个按关键字递增排列的记录，同一文件上的控制区间的大小相同。

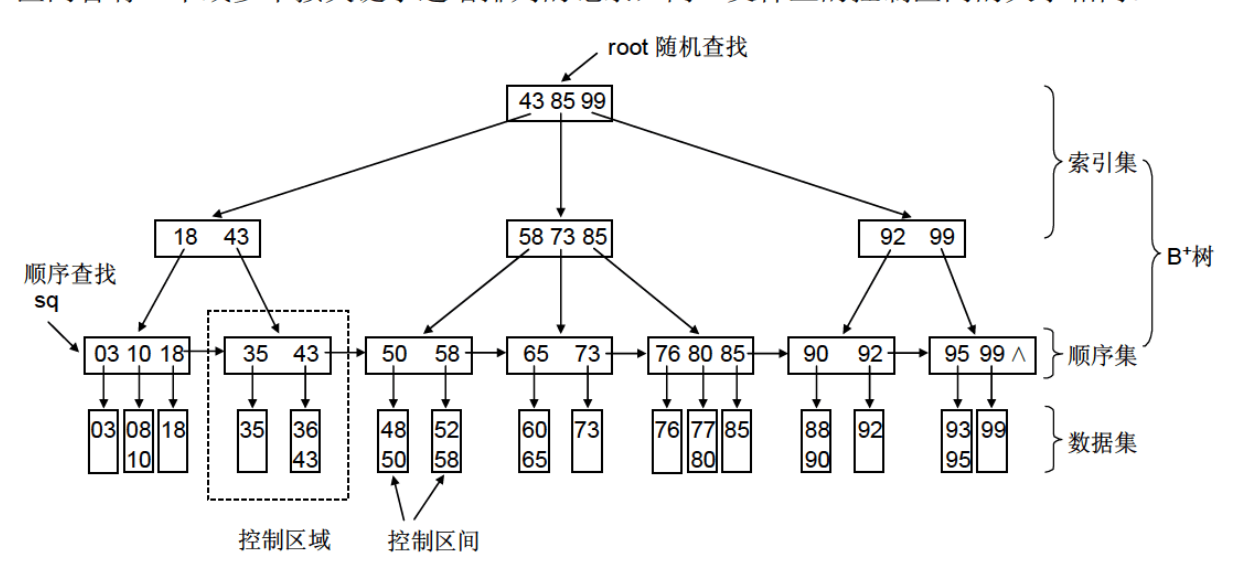


图 9.10　VSAM 文件的结构示意图

索引集和顺序集一起构成一棵 B^+树，它是文件的索引部分。顺序集中存放每个控制区

间的索引项，由两部分信息组成，即该控制区间中的最大关键字和指向控制区间的指针。若干相邻的控制区间的索引项，形成顺序集中的一个结点，结点之间用指针相链接，而每个结点又在其上一层的结点中建有索引，且逐层向上建立索引，所有的索引项都由最大关键字和指针两部分信息组成，这些高层的索引项形成 B^+树的非终端结点。顺序集中一个结点连同其对应的所有控制区间形成一个整体，称做**控制区域**（Control Range）。每个控制区域可视为一个逻辑柱面，而控制区间可视为一个逻辑磁道。

因此，VSAM 文件既可在顺序集中进行顺序存取，又可从最高层的索引（B^+树的根结点）出发，按关键字存取。

在 VSAM 文件中，记录可以是不定长的，因而在控制区间中，除了存放记录本身之外，还有每个记录的控制信息和整个区间的控制信息（如区间中存放的记录数等），控制区间的结构如图 9.11 所示。

记录1	记录2	…	记录n	记录的控制信息	控制区间的控制信息

图 9.11　控制区间的结构示例

VSAM 文件中没有溢出区，解决插入的方法是在初建文件时留出空间：一是每个控制区间内并未填满记录，而是在最末一个记录和控制信息之间留有空隙；二是在每个控制区域中有一些完全空的控制区间，并在顺序集的索引中指明这些空区间。可见，VSAM 文件占用较多的存储空间，一般只能保持平均 75%的存储空间利用率。

当插入新记录时，大多数的新记录能插入到相应的控制区间内，但要注意：为了保持区间内记录的关键字从小至大有序，则需将区间内比待插记录关键字大的记录，向控制信息的方向移动。若控制区间已满，则在下一个记录插入时，要进行控制区间的分裂，即把近乎一半的记录移到同一控制区域内全空的控制区间中，并修改顺序集中相应索引。倘若控制区域中已经没有全空的控制区间，则要进行控制区域的分裂，此时，顺序集中的结点亦要分裂，由此尚需修改索引集中的结点信息。但由于控制区域较大，通常很少发生分裂的情况。

在 VSAM 文件中删除记录时，需将同一控制区间中比待删记录关键字大的记录向前移动，把空间留给以后插入的新记录。若整个控制区间变空，则回收作空闲区间用，且需删除顺序集中相应的索引项。

和 ISAM 文件相比，基于 B^+树的 VSAM 文件有如下优点：能保持较高的查找效率，查找一个后来插入的记录和查找一个原有记录具有相同的速度；动态地分配和释放存储空间；永远不必对文件进行再组织。因而基于 B^+树的 VSAM 文件，通常被作为大型索引顺序文件的标准组织。

9.5 散列文件

散列文件是利用散列技术组织的文件，亦称**直接存取文件**。它类似于散列表，即根据文件中关键字的特点，设计一个散列函数和处理冲突的方法，将记录散列到外存储设备上。

由于存储介质是外存储器，散列文件中的记录通常是成组存放的，这点与散列表不同。若干个记录组成一个存储单位，称做**桶**（Bucket）。假如一个桶能存放 m 个记录，则 m 个互为同义词的记录可以存放在同一地址的桶中，当第 m+1 个同义词出现时才发生“溢出”。处理溢出，也可采用散列表中处理冲突的各种方法，但对散列文件，主要采用拉链法。

当发生“溢出”时，需要将第 m+1 个同义词存放到另一个桶中，通常称此桶为“溢出桶”。相对地，称前 m 个同义词存放的桶为“基桶”。溢出桶和基桶大小相同，相互之间用指针链接。当在基桶中没有找到待查记录时，就沿着指针到所指的溢出桶中进行查找。因此，希望同一散列地址的溢出桶和基桶，在磁盘上的物理位置不要相距太远，最好在同一柱面上。

例如，某文件有 14 个记录，其关键字序列为{09, 19, 01, 14, 22, 25, 33, 27, 26, 05, 15, 18, 29, 12}。桶的容量 m=3，桶数 b=7，用除余法作散列函数 H(key)=key%7。由此得到的散列文件如图 9.12 所示。

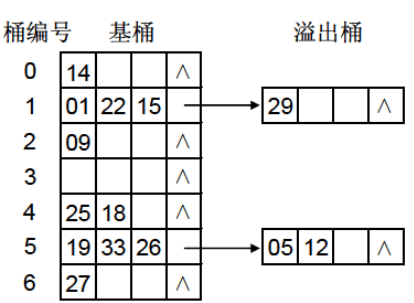


图 9.12　散列文件示例

在散列文件中进行查找时，首先根据给定值求出散列地址（即基桶号），将基桶的记录读入内存，进行顺序查找，若找到关键字等于给定值的记录，则检索成功。当在基桶内查不到时，若基桶没有填满，则文件中不含待查记录，否则根据指针域的值找到溢出桶，并将其中的记录读入内存继续进行顺序查找，直至查找成功或不成功。

在散列文件中删除一个记录，仅需对被删记录作删除标记即可。

散列文件的优点是：文件随机存放，记录不需进行排序；插入、删除方便；存取速度快；不需要索引区，节省存储空间。其缺点是：不能进行顺序存取，只能按关键字随机存取，且查询方式限于精确查询。在经过多次插入、删除后，可能造成文件结构不合理，如溢出桶满而基桶内多数记录已被删除，此时需要重新组织文件。

9.6　多关键字文件

一般地，在对文件进行检索时，不仅要对主关键字进行查询，还经常要对次关键字进行某些查询。例如图 9.1 的职工档案文件，职工号为主关键字，姓名、性别、年龄等为次关键字。对该文件可能要进行如下查询：找 25 岁的男职工。这涉及性别（=“男”）和年龄（=25）两个次关键字。如果文件组织中只有主关键字索引，则这类查询只能顺序存取文件中的每一条记录进行比较，效率很低。为此，除了按以上几节讨论的方法组织文件之外，还需建立一系列的次关键字索引（但次关键字相同的记录一般有多条）。这种包含有若干次关键字索引的文件称为**多关键字文件**。次关键字索引可以是稠密的，也可以是稀疏的；索引表可以是顺序表，也可以是树表。

下面讨论两种多关键字文件的组织方法。

9.6.1 多重表文件

多重表文件（Multilist File）是将索引方法和链接方法相结合的一种组织方式：将相同次关键字的记录构成链表；分别对主关键字和需要查询的次关键字建立主索引和次索引；次索引指向次关键字链表，其索引项由该链表的头指针、链表长度及次关键字组成。其中次关键字链表并不单独建立，而是通过在主文件中增设次关键字指针字段来实现。通常多重表文件的主文件是一个顺序有序文件，从而主索引可建为稀疏索引，但次索引是稠密索引。

注意，头指针和链指针可以是记录的实际物理地址，也可以是记录号，甚至还可以是主关键字。因为主关键字可看成是记录的符号地址，但这样的索引表存取速度较慢，优点是对于存储具有相对独立性。为简单起见，本节以下将索引链取为记录号。

例如图 9.13 是一个多重表文件的示例。主关键字是职工号，次关键字取为性别和年龄。它设有两个链接字段，分别将具有相同性别和相同年龄的记录链在一起，由此形成的性别索引和年龄索引见图 9.13（c）和图 9.13（d）。

记录号	职工号	姓名	性别		年龄	
01	001	区建新	男	03	32	03
02	004	刘　青	女	04	24	04
03	007	谢小刚	男	06	36	∧
04	008	王　红	女	05	26	05
05	010	田　丽	女	∧	28	06
06	012	许志军	男	∧	25	∧

（a）数据文件

主关键字	头指针
007	01
012	04

（b）主索引

次关键字	头指针	长度
男	01	3
女	02	3

（c）“性别”次索引

次关键字	头指针	长度
21～30	02	4
31～40	01	2

（d）“年龄”次索引

图 9.13　多重表文件示例

有了这些索引，便易于处理各种有关次关键字的查询。例如，要查询所有男职工，则只需在性别索引中先找到次关键字“男”的索引项，然后从它的头指针 01 出发，找到 01 号记录，从该记录的性别链中得到下一个是 03 号记录等，可列出该链表上所有的记录。如果是组合查询，如找 25 岁的男职工，则既可以从性别索引中“男”的头指针出发，也可以从年龄索引中“21～30”的头指针出发，读出相应链表上的每个记录，判定它是否满足查询条件。对这种情况，还可先比较这两个链表的长度，然后在较短的链表上查找，可提高查找效率。

在上例中，各个次关键字链表是按主关键字大小链接的。如果不要求保持链表的某种次序，则插入一个新记录是容易的，此时可将记录插在链表的头指针之后。但是，要删去一个记录却很烦琐，需要在每个次关键字的链表中搜索并删去该记录。

9.6.2　倒排文件

倒排文件和多重表文件类似，区别在于次关键字索引的结构不同。倒排文件中的次关键字索引称做**倒排表**。具有相同次关键字的记录之间不进行链接，而是将它们的记录号（或物理地址）直接列在索引表中。例如，对表 9.13 所示的多重表文件，去掉两个链接字段后，所建立的性别倒排表和年龄倒排表，如图 9.14（a）和图 9.14（b）所示。倒排表和主文件一起就构成了**倒排文件**（Inverted File）。

男	01, 03, 06
女	02, 04, 05

（a）“性别”倒排表

21～30	02, 04, 05, 06
31～40	01, 03

（b）“年龄”倒排表

图 9.14　倒排文件索引示例

在图 9.14 的倒排表中，各索引项的记录号是有序的。如果主文件是无序的，则记录号有序并不意味着主关键字有序。这时，还可以将这些记录号按主关键字有序排列。

倒排表的主要优点是查找速度较快：在处理复杂的多关键字查询时，可先在倒排表中完成查询（预查询，包括查询条件的交、并等逻辑运算），得到结果后再对记录进行存取。这样不必对每个记录都随机存取，而把对记录的查询转换为记录号（或地址）集合的运算，从而提高查找速度。例如，要找出所有年龄在 21～30 岁的男职工，则先对年龄为 21～30 的记录集{02, 04, 05, 06}与性别为男的记录集{01, 03, 06}做“交”运算得到{06}，即符合条件的记录为 06 号记录。

在插入和删除记录时，需要修改倒排表。

在一般的文件组织中，是先找记录，然后再找到该记录所含的各次关键字；而倒排文件中，是先给定次关键字，然后查找含有该次关键字的各个记录，这种文件的查找次序正好与一般文件的查找次序相反，因此称之为“倒排”。由此可见，多重表文件实际上也是倒排文件，只不过索引的方法不同。

习　题　九

9.1　什么是文件的逻辑记录和物理记录？它们有什么区别与联系？

9.2　简述磁带和磁盘的结构和存储信息的特点。

9.3　简述 ISAM 文件的组织方法和操作特点。

9.4　简述散列文件的查找方法及优缺点。

9.5　叙述在图 9.10 所示 VSAM 文件 B^+树部分查找键值为 80 的记录的过程。

9.6　文件的组织方法与外存储器的物理特性之间有何联系？试举例说明。

9.7　文件的操作与对内存中数据的操作有何不同？试举例说明。

9.8　在图 9.12 所示的散列文件中，若还有两个键值分别为 39、54 的记录，如何存放？

9.9　文件的检索效率取决于哪些因素？



第 1 章　概论

1.1 解：首先，从数据结构的 3 个方面来看，逻辑结构和运算与计算机无关，它们构成问题的数学模型，即使没有计算机也是存在的。其次，日常生活中的很多问题，如图书、档案等的放置和查找，电话号码、字典等的组织和查找，家谱、人事关系等的记录和查询，多条路径中找最短，多件事情的先后安排等，这些实际上都是数据结构问题。广义地讲，如果将人看成计算机的 CPU、客观空间看成计算机的内外存、要处理的各种事情看成数据，则只要人在活动，数据结构问题就无处不在。

算法只是对特定问题求解步骤的描述，若其实现时不是在计算机上进行，比如手工处理，就不需要转换为计算机程序。

1.2 略

1.3 解：基本存储结构虽然只有 4 种，但可由它们组合出多种非基本储存结构。

1.4 解：计算机性能提高后，原来比较困难的时空复杂度问题也许会有所缓和，但人们往往又会要求处理更大规模和更加复杂的问题。一般计算机性能的提高总难以满足大规模问题的要求，并且问题的规模越大，程序的效率问题就越突出。另外，算法的复杂度越高，则从机器性能提高上得到的收益相对就越小。

通过算法分析，可以在编程前决定某个方案是否可行，以及找到效率问题的关键所在和改进方向。

1.5 解：$(2/3)^n$，2^{100}，$\log_2 n$，$n^{2/3}$，$n\log_2 n$，$n^{3/2}$，$(3/2)^n$，2^n，$n!$，n^n

1.6 解：

（1）循环变量 i 的取值为 i=1, 2, 3, …, k，其中 $k^2 \leqslant n$，则频度 $k \leqslant n^{1/2}$，复杂度为 $O(n^{1/2})$。

（2）这里外循环次数形式上为 n，但内循环从 i^2 到 n，则外循环的实际有效循环次数为 $\sqrt{n}$，故内循环体的总循环次数是

$$\sum_{i=1}^{\sqrt{n}}(n-i^2+1)=\sqrt{n}(n+1)-\sum_{i=1}^{\sqrt{n}}i^2=\sqrt{n}(n+1)-\frac{\sqrt{n}(\sqrt{n}+1)(2\sqrt{n}+1)}{6}=O((\sqrt{n})^3)$$

所以 $T(n)=O(n^{3/2})$。

1.7 解：

（1）主要程序段如下：

```
sum=0;
for(i=1;i<=n;i++)
  sum+=i;
```

显然该程序段频度最大的语句是循环体 sum+=i，它执行了 n 次，所以 T(n)=O(n)。

（2）主要程序段如下：

```
sum=0;
for(i=1;i<=n;i+=2)
  sum+=i;
```

设循环体的执行次数为 k，则最后一个奇数为 2k−1，即有 2k−1≤n，所以 k=O(n)，从而 T(n)=O(n)。实际上，注意到 n 以内的数大约一半为奇数（另一半左右为偶数），则循环体大约执行 n/2 次，也可得到 T(n)=O(n)。

（3）主要程序段如下：

```
sum=0;
for(i=1;sum<n;i++)
  sum+=i;
if(sum>n) sum-=(i-1);        //最后一次累加后可能超过了 n
```

设循环体的执行次数为 k，则 sum=1+2+…+k=k(k+1)/2，欲使 sum≤n，则 $k=O(n^{1/2})$，所以 $T(n)=O(n^{1/2})$。

1.8 解：主要程序段如下：

```
for(i=1;i<=n;i*=2)
  for(j=1;j<=i;j++)
    A[i][j]=0;
```

这里虽然外循环次数为 $\log_2 n$，内循环次数为 i，但内循环体的总循环次数不是 $\sum_{i=1}^{\log_2 n} i = 1+2+3+\cdots+\log_2 n = (\log_2 n)(\log_2 n+1)/2$，因为外循环的循环变量不是每次增 1 而是乘 2，实际执行次数是 $\sum_{k=0}^{\log_2 n} 2^k = 1+2+2^2+\cdots+2^{\log_2 n} = 2^{(\log_2 n)+1}-1 = 2n-1$，所以 T(n)=O(n)。

1.9 解：该题就是在数组 A[n]中找最大值和最小值，显然其比较次数为 2(n−1)次。如果采用如下代码，将这两个过程同时进行：

```
min=max=A[0];
for(i=1;i<n;i++)
  if(A[i]<min) min=A[i];
  else if(A[i]>max) max=A[i];
```

则最好情况是第一个 if 总成立，此时比较次数降为 n−1 次；最坏情况是第一个 if 总不成立，第二个 if 总要执行，比较次数为 2(n−1)次。

1.10 解：一般有两种方法：

（1）加减法：a+=b;b=a−b;a−=b;

（2）异或法：a=a^b;b=a^b;a=a^b;

第 2 章 线性表

2.1 解：{a, b, c, d, e, f}表示集合，（a, b, c, d, e, f）表示线性表。

2.2 略

2.3 略

2.4 解：

（1）这时顺序表的类型定义如下：

```
typedef struct {
  datatype *data;        //线性表数组地址
  int n;                 //线性表当前的长度
} sqlist;
```

除了在初始化时要分配数组空间外，其他运算的算法不变（略）。初始化算法如下：

```
int init(sqlist *L) {
  L->data=new datatype[maxsize];
  if(L->data==NULL) {cout<<"内存不足!\n";return 0;}
  L->n=0;                //修改表长
  return 1;              //初始化成功
}
```

另外要注意，在程序结束时，要释放数组空间。为此可对顺序表增加一个销毁运算：

```
void destroy(sqlist *L) {
  delete[]L->data;
}
```

（2）这时顺序表的类型定义如下：

```
typedef struct {
  datatype data[maxsize+1];     //线性表数组空间，data[0]不用
  int n;                        //线性表当前的长度
} sqlist;
```

除了初始化，其他运算实现时注意元素 a_i 对应的位置为 data[i]即可，具体算法略。

另外，也可用 data[0]来存放数组长度，这时顺序表的类型定义如下：

```
typedef struct {
  datatype data[maxsize+1];     //线性表数组空间，data[0]存放线性表当前的长度
} sqlist;
```

有关算法都要略作修改，具体略。

2.5 解：

（1）采用课本例 2.2 的算法，具体如下：

```
void deletex(sqlist *L,datatype low,datatype high) {
  int i,s;
  s=0;
  for(i=0;i<L->n;i++)
    if(L->data[i]>low && L->data[i]<high) s++; //累计待删结点个数
```

```
    else if(s>0) L->data[i-s]=L->data[i];       //当前点前移 s 个位置
  L->n=L->n-s;                  //调整表长
}
```

（2）注意到表的有序性，先找到待删结点的开始位置，从该处开始累计待删结点个数 s，得到第一个需要前移的结点，将其后所有结点一次性地前移 s 位。算法如下：

```
void deletex(sqlist *L,datatype low,datatype high) {
  int i,s;
  for(i=0;i<L->n;i++)
    if(L->data[i]>low) break;        //得到待删结点的开始位置
  s=0;
  for(;i<L->n;i++)
    if(L->data[i]<high) s++;         //累计待删结点个数
    else break;
  if(s==0) return;                   //没有待删结点
  for(;i<L->n;i++) L->data[i-s]=L->data[i];     //当前点前移 s 个位置
  L->n=L->n-s;              //调整表长
}
```

2.6 解：

（1）对顺序表的每个元素，删除其后值与之相等的所有元素，这可采用题 2.5（1）的方法，算法如下：

```
void purge(sqlist *A) {
  int i,j,s;
  for(i=1;i<A->n;i++) {           //设下标从 1 开始使用
    s=0;
    for(j=i+1;j<=A->n;j++)
      if(A->data[j]==A->data[i]) s++;        //累计重复元素
      else if(s>0) A->data[j-s]=A->data[j]; //前移 s 个位置
    A->n=A->n-s;                  //调整表长
  }
}
```

（2）先找到第一个零元素，然后删除其后所有的零元素。在删除零元素时，采用题 2.5（1）的方法，算法如下：

```
void purge(sqlist *A) {
  int i,j,s;
  for(i=1;i<=A->n;i++)
    if(A->data[i]==0) break;      //找第一个零元素，设下标从 1 开始使用
  s=0;
  for(j=i+1;j<=A->n;j++)
    if(A->data[j]==0) s++;        //累计重复的零元素
    else if(s>0) A->data[j-s]=A->data[j];//前移 s 个位置
  A->n=A->n-s;                    //调整表长
}
```

（3）方法 1：可采用题 2.5（1）的方法，算法略。

方法 2：如果允许改变非零元的相对位置，则也可在删除零元素时，用尾部的非零元素与其交换位置。算法如下：

```
void purge(sqlist *A) {
  int i,j;
```

```
  i=1;j=n;                         //设下标从 1 开始使用
  while(i<j) {
    while(i<j && A->data[i]!=0) i++;     //从前向后找零元素
    while(i<j && A->data[j]==0) j--;     //从后向前找非零元素
    if(i<j) {
      A->data[i]=A->data[j];
      A->data[j]=0;
      i++;j--;
    }
  }
  if(A->data[i]!=0) A->n=i;        //调整表长
  else A->n=i-1;
}
```

2.7 解：

（1）方法 1：将循环右移一位的过程执行 k 次，主要语句如下：

```
for(i=0;i<k;i++) {
  x=A[n];
  for(j=n-1;j>=1;j--) A[j+1]=A[j];
  A[1]=x;
}
```

总移动次数为 k(n+1)。

方法 2：逆置法，问题的要求相当于将数组的前后 2 部分“交换”：AB→BA，可将前后 2 段分别逆置 $AB \to A^rB^r$，再整体逆置：$(A^rB^r)^r \to BA$，也可先整体逆置，再前后两段分别逆置。主要语句如下：

```
for(i=1,j=n-k;i<j;i++,j--) {x=A[i];A[i]=A[j];A[j]=x;} //前 n-k 个元素逆置
for(i=n-k+1,j=n;i<j;i++,j--) {x=A[i];A[i]=A[j];A[j]=x;}//后 k 个元素逆置
for(i=1,j=n;i<j;i++,j--) {x=A[i];A[i]=A[j];A[j]=x;}    //整体逆置
```

交换次数为 $\lfloor (n-k)/2 \rfloor + \lfloor k/2 \rfloor + \lfloor n/2 \rfloor \leqslant n$，总移动次数 $\leqslant 3n$。

方法 3：对数组后 k 个元素中的每个，先空出该位置，再依次将其第 k 前趋，第 k 前趋的第 k 前趋，……，依次后移 k 位，若某前趋为最初空出的位置，则本次前趋搜索结束，继续处理下一个元素；但若总的元素调整次数已达 n，则结束。主要语句如下：

```
num=0;
for(i=n;i>n-k;i--) {
  p=i;
  x=A[p];                    //空出初始位置
  for(q=p 的 k 前趋;q!=i;p=q,q=q 的 k 前趋) {A[p]=A[q];num++;}//累计已处理元素数
  A[p]=x;num++;              //累计已处理元素数
  if(num==n) break;
}
```

将数组空间 A[1..n]看成循环的，则 i 单元的第 k 前趋为(i–1–k+n)%n+1。总移动次数最多为 n+k，最少 n+1（空出初始位置最多 k 次，最少 1 次），在三个方法中是最少的。

以上三个算法隐含 $0 \leqslant k < n$。若 $k \geqslant n$，则算法开始时应加上一句 k=k%n，以去除整圈的循环移位（回到原始位置）。

（2）相当于把数组 AB 循环右移 m 位，可采用（1）题的各种方法，略。

2.8 解：利用有序性，从两个表头开始逐个比较求交：若当前结点值相同则取出，否

则结点值小的后移，直到某个表处理完。算法如下：

（1）以顺序表为存储结构

```
void and_sq(sqlist *A,sqlist *B,sqlist *C) {
  int i,j,k;
  i=j=1;                  //设下标从 1 开始使用
  k=0;
  while(i<=A->n && j<=B->n) {
    if(A->data[i]<B->data[j]) i++;
    else if(A->data[i]>B->data[j]) j++;
    else {
      k++;C->data[k]=A->data[i];
      i++;j++;
    }
  }
  C->n=k;                 //表长
}
```

（2）以单链表为存储结构(设链表带头结点)

```
lklist and_lk(lklist A,lklist B) {
  pointer p,q,s,rear,C;
  C=new node;             //生成头结点
  p=A->next;q=B->next;rear=C;
  while(p!=NULL && q!=NULL) {
    if(p->data<q->data) p=p->next;
    else if(p->data>q->data) q=q->next;
    else {
      s=new node;
      s->data=p->data;rear->next=s;rear=s;  //尾插法建表
      p=p->next;q=q->next;
    }
  }
  rear->next=NULL;            //表尾置空
  return C;
}
```

2.9 解：该题与例 2.5 相同，可参见图 2.22，在链表搜索过程中进行结点的判断和删除。搜索过程中，随时记住当前结点*p 的前趋*q，算法如下：

```
void deletex(lklist head,datatype low,datatype high) {
  pointer p,q;
  q=head;
  while(p=q->next,p!=NULL)
    if(p->data>low && p->data<high) {
      q->next=p->next;
      delete p;
    }
    else {
      q=p;
    }
}
```

其中，每次 p 取新值的语句写在了 while 条件中。

2.10 解：合并过程就是每次将两个链表当前结点中较小的一个加入到新链表，如果某个链表先处理完，则将另一个链表的剩下部分直接加入到新链表。将原两个链表头结点之

一作为新链表的头结点，采用尾插法建表。

（1）设单链表有头结点，合并算法如下：

```
lklist merge(lklist A,lklist B) {
  pointer p,q,rear;
  rear=A;                          //取头结点 A 作为新链表的头结点
  p=A->next;q=B->next;
  while(p!=NULL && q!=NULL)
    if(p->data<q->data) {          //将 A 表结点加入到新表
      rear->next=p;rear=p;
      p=p->next;
    }
    else {
      rear->next=q;rear=q;
      q=q->next;
    }
  if(p!=NULL) rear->next=p;        //原 A 链表还有剩下的结点
  if(q!=NULL) rear->next=q;        //原 B 链表还有剩下的结点
  return A;
}
```

（2）设单链表无头结点，合并算法如下：

```
lklist merge(lklist A,lklist B) {
  pointer C,p,q,rear;
  if(A==NULL) return B;            //A、B 链表有一个为空时
  if(B==NULL) return A;
  if(A->data<=B->data) {C=A;p=A->next;q=B;}//A、B 中较小者为新链表 C 的首结点
  else                {C=B;p=A;q=B->next;}
  rear=C;
  while(p!=NULL && q!=NULL)
    if(p->data<q->data) {          //将 A 链表结点加入到新表
      rear->next=p;rear=p;
      p=p->next;
    }
    else {
      rear->next=q;rear=q;
      q=q->next;
    }
  if(p!=NULL) rear->next=p;        //A、B 链表之一还有剩下的结点
  if(q!=NULL) rear->next=q;
  return C;
}
```

本题也可这样求解：以两个链表中的一个作基础，将另一个链表中的结点逐个删除并插入进来。由于是就地进行，结点的删除和插入仅对有关指针重新进行链接。算法略。

2.11 解：设单链表有头结点。

（1）算法思想：从第 2 个结点开始判断当前结点是否比其前趋大，算法如下：

```
int detect(lklist L) {
  pointer p,q;                        //p 指向当前结点，q 为其前趋
  p=L->next;if(p==NULL) return 1;     //链表为空
  while(q=p,p=p->next,p!=NULL)
```

```
    if(p->data<q->data) return 0;   //不满足递增
  return 1;
}
```

其中，向后推进的语句写在了 while 条件中。

（2）算法思想：先求前 2 个结点的差 d，然后从第 3 个结点开始判断当前结点与其前趋的差是否等于 d，算法如下：

```
int detect(lklist L) {
  pointer p,q;                      //p 指当前结点，q 为其前趋
  datatype d;
  q=L->next;if(q==NULL) return 1;   //链表为空
  p=q->next;if(p==NULL) return 1;   //链表只有 1 个结点
  d=p->data-q->data;
  while(q=p,p=p->next,p!=NULL)
    if(p->data-q->data!=d) return 0;//不满足等差
  return 1;
}
```

（3）算法思想：先求前 2 个结点的比 d，然后从第 3 个结点开始判断当前结点与其前趋的比是否等于 d。算法与上题类似，略。

2.12 解：设多项式用循环单链表表示（设其中各结点按指数递减排列）。多项式相加的规则是，对两个多项式链表的当前结点，如果指数相同，则将系数相加，若该系数和不为 0，则生成新多项式链表的结点；如果指数不同，则按指数较大的项生成新链表的结点；如果某个多项式先处理完，则按另一个链表的剩余部分逐个生成新链表的结点。设相加的两个多项式链表为 A 和 B，非形式算法如下：

```
生成新链表头结点*C;
新链表尾指针 rear 指向头结点*C;          //空表
用 p 和 q 分别指向多项式链表 A 和 B 的首结点;
while(A 链表和 B 链表都没有处理完) {
  if(当前结点指数相同) {
    计算系数和 x;
    if(x 不为零) {生成新结点;尾插到新链表;按 x 赋值新结点;A、B 当前结点都前进一步;}
  }
  else {
    生成新结点;尾插到新链表;
    if(A 结点指数大于 B 结点) {按*p 赋值新结点;A 当前结点前进一步;}
    else                     {按*q 赋值新结点;B 当前结点前进一步;}
  }
}
whlle(A 链表没有处理完) {
  生成新结点;尾插到新链表;
  按*p 赋值新结点;A 当前结点前进一步;
}
while(B 链表没有处理完) {
  生成新结点;尾插到新链表;
  按*q 赋值新结点;B 当前结点前进一步;
}
新链表尾结点后继指向头结点;              //使成循环链表
返回新链表头指针 C;
```

依据该算法就不难写出具体的C/C++语言算法了，如判断A链表是否处理完的条件为p!=A，A链表当前结点前进一步的语句为p=p->next等，具体算法略。

第3章　栈、队列和串

3.1 解：由出栈序列模拟进出栈过程，可知其间栈内最多同时有3个元素，故栈的容量至少为3。

3.2

（1）证：这是显然的：i，j，k依次入栈，但k最先出栈，则它之前入栈的i和j还在栈中，当k出栈后，因j比i更靠近栈顶，以后j应比i先出栈。

（2）*证一：考察第1个入栈数据“1”，它在以后可能是第1、2、…、n个出栈。不妨设为第i个出栈，则其后的i–1个数“2”、“3”、…、“i”要完成进出栈后它才可出栈；而它出栈后，剩下的n–i个数“i+1”～“n”也要完成进出栈。设n个结点的出栈序列数为b_n，则前i–1个数的出栈序列数为b_{i-1}，后n–i个数的出栈序列数为b_{n-i}，故“1”为第i个出栈的出栈序列数为$b_{i-1}b_{n-i}$。但i可以是{1, 2, …, n}中的任一个，故总的序列数为$\sum_{i=1}^{n} b_{i-1}b_{n-i}$。不妨设$b_0$=1，故得递推式：

$$\begin{cases} b_0 = 1 \\ b_n = \sum_{i=1}^{n} b_{i-1}b_{n-i} & n \geqslant 1 \end{cases}$$

通式为$b_n = \frac{1}{n+1} \cdot \frac{(2n)!}{n!n!} = \frac{1}{n+1}C_{2n}^{n}$（见附录E），前几项为1, 2, 5, 14, 42, 132, 429,…。

证二：将进、出栈分别用“1”、“0”表示，则n个数据的进出栈过程可表示为2n位二进制数，其中“1”和“0”各占n位。

在2n位中填入n个1（其余填0）的方案数为C_{2n}^{n}，其中有些不符合要求（从左向右扫描，遇到0的个数比1的个数多时），减去这些不符合要求的方案数即为所求。

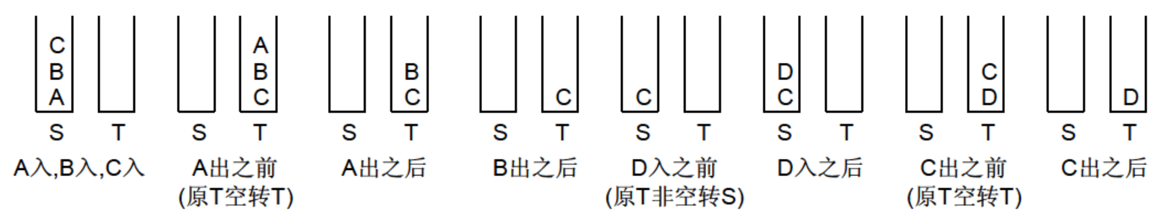
设序列在某位k时不合要求了，即此时0比1多1个，则其后面1比0多1个。若把后面的0和1互换，则整个序列为n+1个0和n−1个1组成的2n位数，即一个不合要求的方案对应于一个由n+1个0和n−1个1组成的2n位数。

反之，任何一个由n+1个0和n−1个1组成的2n位数，从左向右扫描，必在某个位置k时0的个数比1的个数多（一定会出现，至少因为0比1多）。同样，此时把其后面的0和1互换，则得到由n个0和n个1组成的2n位数，即n+1个0和n−1个1组成的2n位数必对应一个不符合要求的方案。

因而不合要求的方案数与n+1个0和n−1个1组成的2n位数一一对应，这些数有C_{2n}^{n+1}个（对0而言）或C_{2n}^{n-1}个（对1而言，二者相等），所以有效的输出序列的总数$=C_{2n}^{n} - C_{2n}^{n-1} = \frac{1}{n+1}C_{2n}^{n}$。

3.3 解：栈的特点是先进后出，队列的特点是先进先出。如果用两个栈，一个栈S先进后出，另一个栈T后进先出，两者组合则可实现先进先出。入队时，若栈T非空，则先

将 T 中元素反入栈到 S 中，再将输入数据入栈 S；出队时，若栈 T 非空，则从 T 中出栈，否则先将 S 元素依次出栈并入栈 T，再从 T 中出栈。以 A 入、B 入、C 入、A 出、B 出、D 入、C 出的队列顺序为例，栈 S 和 T 的变化如下图所示。



通俗地讲，一个栈不能先进先出，为达目的，用另一个栈把数据“倒腾”一下。

3.4 解：共享顺序栈的类型定义和进出栈算法如下：

```
typedef struct {
  datatype data[maxsize];
  int top[2];
} sqstack;
void init(sqstack *S) {
  S->top[0]=-1;
  S->top[1]=maxsize;
}
int push(sqstack *S,datatype x,int k) {
  if(S->top[0]+1==S->top[1]) {cout<<"栈满，不能进栈！\n";return 0;}  //上溢
  if(k==0) S->top[k]++;          //改栈顶指针，加 1 或减 1
  else     S->top[k]--;
  sq->data[S->top[k]]=x;         //将 x 插入当前栈顶
  return 1;
}
int pop(sqstack *S,datatype *x,int k) {//栈顶元素由参数返回
  if((k==0 && S->top[k]==-1) || (k==1 && S->top[k]==maxsize))
    {cout<<"栈空，不能退栈！\n";return 0;}//下溢
  *x=sq->data[S->top[k]];        //取出栈顶元素值给 x
  if(k==0) S->top[k]--;          //改栈顶指针，减 1 或加 1
  else     S->top[k]++;
  return 1;
}
```

3.5 解：当数据为负时直接输出，若为正则进栈。当输入数据处理完后，栈内都为正数，将其依次出栈输出即可。假设输入数据为整数，并存放在数组中，则算法如下：

```
void order(int A[],int n) {
  int i,x;
  sqstack S;                        //假设采用顺序栈
  init_sqstack(&S);
  for(i=0;i<n;i++)
    if(A[i]>0) push_sqstack(&S,A[i]);
    else cout<<A[i]<<"  ";
  while(!empty_sqstack(S)) {
    pop_sqstack(&S,&x);
    cout<<x<<"  ";
  }
  cout<<"\n";
}
```

3.6 解：

方法 1：对表达式扫描，遇到左括号“(”就进栈，遇到右括号“)”就退栈。若中途出现栈空又要退栈则右括号有多余；若结束时栈非空，则左括号有多余。算法略。

方法 2：设置括号计数器 n，初值为 0。扫描表达式，遇到“(”则 n++，遇到“)”则 n−−。若中途出现 n<0 则右括号有多余；若结束时 n>0，则左括号有多余。算法略。

3.7 解：将单链表各元素依次进栈，再依次出栈。假设链表没有头结点，算法如下：

```
void inverse(lklist head) {
  pointer p;
  sqstack S;
  init_sqstack(&S);
  p=head;
  while(p!=NULL) {
    push_sqstack(&S,p->data);
    p=p->next;
  }
  p=head;
  while(!empty_sqstack(&S)) {
    pop_sqstack(&S,&p->data);
    p=p->next;
  }
}
```

3.8 解：

方法 1：先将链表的前一半字符依次进栈，然后依次出栈与链表的后一半字符依次比较（当字符数为奇数时，中间位置上的字符不入栈也不比较）。算法略。

方法 2：先将链表的所有字符依次进栈，然后将后一半的字符依次出栈与链表从头开始的前一半字符依次比较。算法略。

方法 3：不用辅助栈，先将链表的前一半字符就地逆置，再依次与链表的后一半字符依次比较。判断结束后，再将链表的前一半字符逆置回去。算法略。

3.9* 解：一些简单的递归可以转换为循环（见第 5 章），反之，一些简单的循环也可以转换为递归。这里的关键是将问题写成递归形式。设 $f(n)=1+2+3+\cdots+n$，则显然 $f(n)=f(n-1)+n$，且 $f(1)=1$，于是得到递归算法如下：

```
int sum(int n) {
  if(n==1) return 1;
  return f(n-1)+n;
}
```

3.10 解：这时头指针为 front=rear−>next，队空的条件为 rear−>next=rear。出队时采用等效算法，删除原头结点，使原首结点成为头结点。算法如下：

```
void enqueue(lklist rear,datatype x) {
  pointer p;
  p=new node;              //申请新结点空间
  p->data=x;               //给新结点赋值
  p->next=rear->next;      //新结点 next 指向头结点
  rear->next=p;            //原尾指针指向新结点
  rear=p;                  //新结点成为新尾结点
}
int dequeue(lklist rear,datatype *x) {
  pointer s;
  if(rear->next==rear) {cout<<"队空，不能出队！\n";return 0;}//队空下溢
```

```
    s=rear->next;              //s 指向头结点
    *x=s->next->data;          //取出原队头数据
    rear->next=s->next;        //头指针指向原队头
    delete s;                  //释放原头结点
    return 1;
  }
```

3.11 解：注意到队列的循环性，利用取模运算，可得三者之间的关系为：

$$len=(rear-front+m)\%m$$
$$rear=(front+len)\%m$$
$$front=(rear-len+m)\%m$$

3.12 解：

（1）这里需对出队算法进行修改，即出队时先计算队头指针 front=(rear−len+m)%m，然后再循环加 1，这两句可合写为 front=(rear−len+m+1)%m。另外，由于已知队列长度，所以判断上下溢的条件也要修改。算法如下：

```
  int en_sqqueue(sqqueue *sq,datatype x) {
    if((sq->len==m) {prinf("队满，不能入队！\n");return 0;}//队满上溢
    sq->rear=(sq->rear+1)%m;
    sq->data[sq->rear]=x;
    return 1;
  }
  int de_sqqueue(sqqueue *sq,datatype *x) {       //出队，由 x 返回原队头值
    int front;
    if(sq->len==0) {cout<<"队空，不能出队！\n";return 0;}   //队空下溢
    front=(sq->rear-sq->len+m+1)%m;
    *x=sq->data[front];
    return 1;
  }
```

（2）与上题类似，这里要修改入队算法，即入队时先计算队尾指针 rear=(front+len)%m，然后再循环加 1，这两句可合写为 rear=(front+len+1)%m。同样，由于已知队列长度，所以判断上下溢的条件也要修改。算法如下：

```
  int en_sqqueue(sqqueue *sq,datatype x) {
    int rear;
    if((sq->len==m) {prinf("队满，不能入队！\n");return 0;}//队满上溢
    rear=(sq->front+sq->len+1)%m;
    sq->data[rear]=x;
    return 1;
  }
  int de_sqqueue(sqqueue *sq,datatype *x) {   //出队，由 x 返回原队头值
    if(sq->len==0) {cout<<"队空，不能出队！\n";return 0;}   //队空下溢
    sq->front=(sq->front+1)%m;
    *x=sq->data[sq->front];
    return 1;
  }
```

（3）标志位的含义不同时，算法会有所不同。以下假设标志位定义如下：若入队后出现 rear=front，则置 flag=1；其他情况 flag=0。于是队空条件为 rear=front 且 flag=0，队满条件为 flag=1。算法如下：

```
int en_sqqueue(sqqueue *sq,datatype x) {
  if(sq->flag==1) {prinf("队满，不能入队! \n");return 0;}//队满上溢
  sq->rear=(sq->rear+1)%m;
  sq->data[sq->rear]=x;
  if(sq->rear==sq->front) sq->flag=1;
  return 1;
}
int de_sqqueue(sqqueue *sq,datatype *x) {   //出队，由 x 返回原队头值
  if(sq->rear==sq->front && sq->flag==0)
   {cout<<"队空，不能出队! \n";return 0;}     //队空下溢
  sq->front=(sq->front+1)%m;
  *x=sq->data[sq->front];
  sq->flag=0;
  return 1;
}
```

也可用 flag=1 表示空，其他情况 flag=0；或者 flag=1 表示满，flag=−1 表示空，其他情况 flag=0 表示既不空也不满等，上述算法要相应地进行调整。

3.13 解：这里循环队列元素的下标范围是 1 到 m，即当 front 和 rear 当前位置为 m 时，入队或出队后下一个位置应为 1，这时循环意义的加 1 语句为：i=i%m+1。

（1）初始化：rear=front=1(可为 1 到 m 中任一值)

（2）入队：rear=rear%m+1; data[rear]=x;

（3）出队：front=front%m+1; x=data[front];

（4）队空：rear=front

（5）队满：rear%m+1=front

3.14 解：这时如果 front=rear，则队列中只有 1 个元素；若再出队则队空，此时 front 比 rear 大 1（循环意义下），但显然队满时 front 也比 rear 大 1，即该条件不能区分队空和队满。这里采用课本的方法，即当队列仅剩一个单元时认为队满，以便区分队空和队满。于是：

队空：front=(rear+1)%m

队满：front=(rear+2)%m

初始化：rear=0; front=1;

入队：rear=(rear+1)%m; data[rear]=x;

出队：x=data[front]; front=(front+1)%m;

长度：len=(rear−front+1+m)%m

3.15 解：操作过程如下：

（1）将栈中所有元素依次出栈、入队，则栈空，队列为 $\{B_1,B_2,\cdots,B_n,A_n,A_{n-1},\cdots,A_1\}$。

（2）将 $\{B_1,B_2,\cdots,B_n\}$ 依次出队、入队，则队列为 $\{A_n,A_{n-1},\cdots,A_1,B_1,B_2,\cdots,B_n\}$。

（3）将 $\{A_n,A_{n-1},\cdots,A_1\}$ 出队、入栈，则栈为 $\{A_n,A_{n-1},\cdots,A_1$ (栈顶)$\}$，队列为 $\{B_1, B_2,\cdots,B_n\}$。

（4）依次 B_i 出队、入队，A_i 出栈、入队，则最后为 $\{B_1,A_1,B_2,A_2,B_3,A_3,\cdots,B_n,A_n\}$。

总的运算量为 2n+2n+2n+4n=10n。

3.16 略

3.17 解：（1）两者相同；（2）两者之一为空串；（3）其中一个为另一个的重复串。

3.18 解：假设串结束符为“\0”，算法如下：

```
void concat(char s1[],char s2[]) {
  int i,j;
  i=0;
  while(s1[i]!='\0') i++;               //找到串 s1 的尾部
  j=0;
  while(s2[j]!='\0') s1[i++]=s2[j++];   //串 s2 连接到 s1 的尾部
  s1[i]='\0';                           //置结束符
}
```

3.19 解：假设串结束符为“\0”，算法如下：

```
void copy(char s1[],char s2[]) {
  int j;
  j=0;
  while(s2[j]!='\0') s1[j]=s2[j++];     //串 s2 拷贝到 s1
  s1[j]='\0';                           //置结束符
}
```

3.20 解：依次取出 S 的每个字符，然后与 T 的所有字符进行比较，设链表不带头结点，算法如下：

```
pointer find(lkstring S,lkstring T) {
  pointer p,q;
  p=S;
  while(p!=NULL) {
    q=T;
    while(q!=NULL && q->ch!=p->ch) q=q->next;
    if(q==NULL) return p;               //T 中没有 S 的当前字符
    p=p->next;
  }
  return NULL;                          //S 的所有字符均在 T 中出现
}
```

3.21* 解：这时数组从下标 1 开始存放字符串，不妨将串元素序号也从 1 开始，则类似从 0 开始时的推导可得：设比较失败时主串、模式串位置分别为 i、j，即 $s_i \neq t_j$，之前“$t_1t_2\cdots t_{j-1}$” = “$s_{i'}s_{i'+1}\cdots s_{i-1}$”。若 t_j 之前模式串存在 “$t_1t_2\cdots t_{k-1}$” = “$t_pt_{p+1}\cdots t_{j-1}$” 的首尾重复子串，则主串可从当前位置 s_i 继续和模式串的 t_k 比较，仍记 next[j]=k，这时 next[j]的定义为：

$$next[j]=\begin{cases}0 & \text{当}j=1\text{时}\\ \max\{k|1<k<j\text{ 且 “}t_1\cdots t_{k-1}\text{”=“}t_{j-k+1}\cdots t_{j-1}\text{”}\} & \text{当此集合非空时}\\ 1 & \text{其他情况}\end{cases}$$

注意这时 k 的含义为首尾重复子串的长度（=k−1）加 1。KMP 算法如下：

```
#define MMAX 100               //MMAX 为串长上限(不能超过 1 字节最大整数 255)
typedef char sstring[MMAX+1];//0 号单元存放串长(不能超过 MMAX)
void NEXT1(sstring *t,int next[]) {//求 next，串从 t[1]开始
  int j,k;
  j=1;k=0;next[1]=0;
  while(j<t[0]) {                    //t[0]存放串长
    if(k==0 || t[j]==t[k]) {
      j++;k++;
      next[j]=k;
    }
```

```
        else k=next[k];
      }
    }
    int KMP1(sstring *s,sstring *t) {//KMP 算法，串从 s[1]、t[1]开始
      int i,j;
      int next[MMAX+1];
      NEXT1(t,next);
      i=1,j=1;
      while(i<=s[0] && j<=t[0]) {      //s[0]、t[0]存放串长
        if(j==0 || s[i]==t[j]) {i++;j++;}
        else  j=next[j];
      }
      if(j>t[0]) return i-t[0];
      else return 0;
    }
```

3.22* 解：有 2 种情况（两者差 1，但实际效果相同）：

	串元素序号从 0 开始											串元素序号从 1 开始										
j	0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	11
模式 t	a	b	c	a	a	c	a	b	a	c	a	a	b	c	a	a	c	a	b	a	c	a
next[j]	–1	0	0	0	1	1	0	1	2	1	0	0	1	1	1	2	2	1	2	3	2	1
nextval[j]	–1	0	0	–1	1	1	–1	0	2	1	–1	0	1	1	0	2	2	0	1	3	2	0

其中也可令前者 next[0]=0，后者 next[1]=1，这时需对 KMP 算法语句略作调整。

第 4 章　多维数组和广义表

4.1 解：按行序排列时先变右边的下标，结果为：

A[-1][1][3]、A[-1][1][4]、A[-1][2][3]、A[-1][2][4]、A[0][1][3]、A[0][1][4]、A[0][2][3]、A[0][2][4]

按列序排列时先变左边的下标，结果为

A[-1][1][3]、A[0][1][3]、A[-1][2][3]、A[0][2][3]、A[-1][1][4]、A[0][1][4]、A[-1][2][4]、A[0][2][4]

4.2 解：按行存储时，从数组开始元素 A[r1][s1][t1]到 A[r][s][t]的元素个数为 $k=(r-r1)\times(s2-s1+1)(t2-t1+1)+(s-s1)\times(t2-t1+1)+(t-t1+1)$，所求地址为：

loc(r,s,t)=loc(r1,s1,t1)+(k-1)×c

按列存储时，从数组开始元素 A[r1][s1][t1] 到 A[r][s][t] 的元素个数为 $k=(t-t1)\times(s2-s1+1)(r2-r1+1)+(s-s1)\times(r2-r1+1)+(r-r1+1)$，所求地址为：

loc(r,s,t)=loc(r1,s1,t1)+(k-1)×c

4.3 解：上三角部分的前 i–1 行元素个数为 $n+(n-1)+(n-2)+\cdots+(n-i+2)$，第 i 行从对角元素到第 j 列元素个数为 j–i+1，所以

$k=[n+(n-1)+(n-2)+\cdots+(n-i+2)]+(j-i+1)=i[n-(i-1)/2]+j-n$

4.4 解：设行号为 i，列号为 j，则：

$$k=i(i-1)/2+j \qquad (1\leqslant j\leqslant i)$$

注意 j 的范围 $1\leqslant j\leqslant i$，由上式得

$i(i-1)/2+1\leqslant k\leqslant i(i-1)/2+i=i(i+1)/2$，即 $i^2-i-2k+2\leqslant 0$ 且 $i^2+i-2k\geqslant 0$

解该不等式组，得

$$\frac{\sqrt{1+8k}-1}{2}\leqslant i\leqslant \frac{\sqrt{1+8k-8}+1}{2}<\frac{\sqrt{1+8k}+1}{2}=\frac{\sqrt{1+8k}-1}{2}+1$$

即

$$i=\left\lceil \frac{\sqrt{1+8k}-1}{2}\right\rceil$$

求出 i 后便可得 $j=k-i(i-1)/2$。

4.5 解：设三元组表按行序存放。算法中先找第 i 行开始位置，再找第 j 列。算法如下：

```
datatype get(spmatrix *A,int i,int j) {
  int p;
  p=1;                                    //设三元组表从下标 1 开始使用
  while(p<=A->t && A->data[p].i<i) p++;   //找第 i 行
  if(p>A->t || A->data[p].i!=i) return 0; //无第 i 行元素
  while(p<=A->t && A->data[p].i==i && A->data[p].j<j) p++; //找第 j 列
  if(p>A->t || A->data[p].j!=j) return 0; //无第 j 列元素
  else return A->data[p].val;
}
```

以上查找的条件取为 data[p].i<i 和 data[p].j<j，而不是 data[p].i!=i 和 data[p].j!=j，这是利用了三元组按行序排列的特点，即元素按行号递增排列，同行的元素按列号递增排列。这样可使在查找到行号或列号大于给定值时提前结束而不必扫描完所有三元组。

为了避免查找过程中对三元组下标范围的检查，可以采用“监视哨”技术，即查找前先在 t+1 处存入(i, j, x)，其中 x 任意。则不论是对行还是对列的查找，最多结束于 t+1 处。算法如下：

```
datatype get(spmatrix *A,int i,int j) {
  int p;
  A->data[A->t+1].i=i;
  A->data[A->t+1].j=j;
  p=1;
  while(A->data[p].i<i) p++;              //找第 i 行
  if(p>A->t || A->data[p].i>i) return 0;  //无第 i 行元素
  while(A->data[p].i==i && A->data[p].j<j) p++;    //找第 j 列
  if(p>A->t || A->data[p].j>j) return 0;  //无第 j 列元素
  else return A->data[p].val;
}
```

如果对三元组表从后向前搜索，则把“监视哨”设在 0 号单元，以上算法需略做修改。

4.6 可借鉴题 4.5 的方法求各元素的位置，具体略。

4.7 注意下三角压缩存储时元素地址的对应关系即可，具体略。

4.8 注意按行存储时元素地址的对应关系即可，具体略。

4.9 略

4.10 解：

（1）取表尾 A=tail(L)：得到((there, (the, here), where), which, what)。

（2）取表头 B=head(A)：得到(there, (the, here), where)。

（3）取表尾 C=tail(B)：得到((the, here), where)。

（4）取表头 D=head(C)：得到(the, here)。

（5）取表尾 E=tail(D)：得到(here)。

（6）取表头 F=head(E)：得到 here。

总的过程为：head(tail(head(tail(head(tail(L))))))

4.11 解：长度=4、深度=3、表头=(this)、表尾见上题（1）。

第 5 章　树形结构

5.1 解：二叉树每个结点最多两个孩子，即结点的度最大为 2，但也可能所有结点的度都不为 2，如只有 1 个结点，或单枝树等，所以二叉树的度并不一定是 2。

5.2 解：三叉树中结点总数 n 等于各种度数的结点数之和：

$$n=n_0+n_1+n_2+n_3$$

另一方面，由于 i 度结点有 i 个孩子，故三叉树中孩子结点数为 $n_1+2n_2+3n_3$，但根不是任何结点的孩子，故三叉树结点总数又可表示为孩子结点数和根之和：

$$n=n_1+2n_2+3n_3+1$$

上面两式相减，即得叶子数 $n_0=n_2+2n_3+1$。

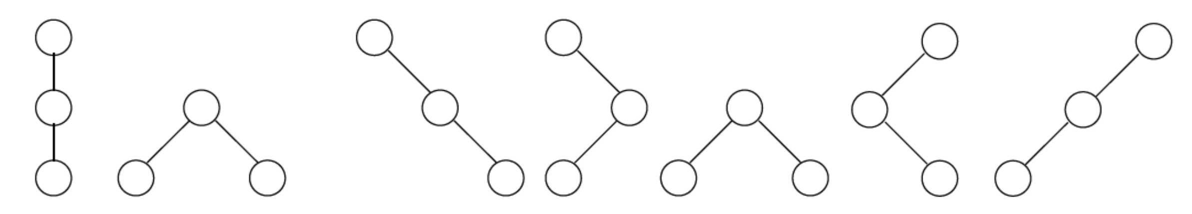
5.3 解：设叶子结点有 n_0 个，则非叶子结点（即分支结点）有 $n-n_0$ 个，每个分支对应一个孩子，故孩子总数为 $(n-n_0)k$ 个，但树中除根外其余结点都是孩子，有 $n-1$ 个，所以 $n-1=(n-n_0)k$，从中可得 $n_0=n-(n-1)/k$。

特别地，满二叉树中叶子数为 $n-(n-1)/2=(n+1)/2$。

5.4 解：这种二叉树只要每一层一个结点即可，故可有多棵。相当于在高度为 n 的满二叉树上沿分支关系每一层取 1 个结点，得到的是根到叶子的路径，所以这种二叉树的个数就是满二叉树的叶子数。高度为 n 的满二叉树有 2^{n-1} 个叶子，即这种二叉树有 2^{n-1} 个。

5.5 解：

（1）注意树的子树不分左右，而二叉树的子树分左右，所以 3 个结点的树只有 2 种形态，3 个结点的二叉树有 5 种形态，见下图。



（a）3 结点树的 2 种形态　　（b）3 结点二叉树的 5 种形态

每种形态下 3 个结点的不同排列有 3!=6 种，故 3 个结点的树和二叉树分别有 2×6=12 和 5×6=30 棵不同的树。

（2）对二叉树可递推求。设 n 个结点的二叉树的形态数为 b_n。若左子树结点数为 i，则右子树结点数为 $n-i-1$，于是二叉树形态数为 b_ib_{n-i-1}。但 i 可以是 0, 1, 2, ⋯, n−1 中的任一个，故总的形态数为 $\sum_{i=0}^{n-1} b_i b_{n-i-1}$。显然，n=0 时形态数为 1（即空树），故得递推式：

$$\begin{cases} b_0 = 1 \\ b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1} & n \geqslant 1 \end{cases}$$

通式 $b_n = \frac{1}{n+1} \cdot \frac{(2n)!}{n!n!} = \frac{1}{n+1} C_{2n}^n$（见附录 E），前几项为{1, 2, 5, 14, 42, …}，增长很快。

这个结果与给定入栈顺序时，所有可能的出栈次序数相同（习题 3.2）。实际上，二叉树的中根遍历过程就是一系列结点的进出栈过程，因此，假设入栈顺序就是二叉树的先根序列，则所有可能的出栈次序就是所有可能的中根序列，而二叉树由先根序列和中根序列决定，所以二叉树的形态数和可能的出栈次序数二者是一致的。

对于树，可想象先将其转换成二叉树，由于转换后二叉树没有右子树，所以 n 个结点树的形态数 t_n 和 n−1 个结点二叉树的形态数 b_{n-1} 相同，即 $t_n = b_{n-1}$。

5.6 解：

（1）先序序列为 DLR，中序序列为 LDR，二者相同则需没有 L，即任一点都没有左子树，该二叉树为右单支树。另外，空树也满足条件。

（2）后序序列为 LRD，中序序列为 LDR，二者相同则需没有 R，即任一点都没有右子树，该二叉树为左单支树。另外，空树也满足条件。

（3）先序序列为 DLR，后序序列为 LRD，二者相同则需没有 L 和 R，即二叉树只有一个结点。另外，空树也满足条件。

（4）先序序列为 DLR，中序序列为 LDR，后序序列为 LRD，三者相同则需没有 L 和 R，即二叉树只有一个结点。另外，空树也满足条件。

（5）先序序列为 DLR，后序序列为 LRD，二者相反则需 L 和 R 中有一个没有，或两个都没有，即任一点只有一个子树（单支树），或仅含一个结点（它既是根又是叶子），它们的共同特点是树中只有一个叶子。

5.7 解：

（1）中序序列为 LDR，先序序列为 DLR，显然，若无 R 但有 L，则中序序列的最后一个结点是根 R，而先序序列的最后一个结点在 L 中，显然结论不对。易见，若中序序列的最后一个结点是叶子的话，结论正确。

（2）后根序列为 LRD，逆序后为 DRL，即按根、右子树、左子树的顺序遍历即可。

5.8 解：

（1）考虑完全二叉树的层序编号。编号为 i 的结点若为叶子，则没有左孩子，即 $2i>n$，所以 $i>n/2$ 的结点都为叶子，或者说 $i \leqslant n/2$ 的结点为非叶子，所以非叶子结点有 $\lfloor n/2 \rfloor$ 个，叶子结点有 $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$ 个，即叶子和非叶子结点都有一半左右。

（2）严格二叉树中没有度为 1 的结点，所以 $n=n_0+n_2$，而 $n_0=n_2+1$，此二式联立即得 $n_0=(n+1)/2$。注意，对这类二叉树，$n_0=n_2+1$ 就是说叶子比非叶子结点数多 1。

特别地，满二叉树既是完全二叉树，又是严格二叉树，所以其叶子数也为 $n_0=(n+1)/2$。

5.9 证明：

（1）设结点总数为 m，由上题有 $n=\lceil m/2 \rceil$，若 m 为偶数，则 $n=\lceil m/2 \rceil = m/2$，所以 $m=2n$；若 m 为奇数，则 $n=\lceil m/2 \rceil = (m+1)/2$，所以 $m=2n-1$。

或证：由于 $m=n_0+n_1+n_2$，而 $n_0=n_2+1$，所以 $m=2n_0+n_1-1$。而完全二叉树度为 1 的结点数要么为 0，要么为 1，所以 $m=2n_0$ 或 $2n_0-1$。

由此题可知，叶子数一定的完全二叉树可有两棵。

（2）设其深度为 k，则当其为满二叉树时，叶子数最多，都在最底层，为 2^{k-1}；当其倒数第二层叶子数最多，最底层只有 1 个叶子时，叶子总数最少，为 $(2^{k-2}-1)+1=2^{k-2}$。所以：

$2^{k-2}\leqslant n\leqslant 2^{k-1}$，即 $k-2\leqslant \log_2 n \leqslant k-1$，或 $\log_2 n+1\leqslant k\leqslant \log_2 n+2$

由于 k 为整数，所以 $k=\lceil \log_2 n+1\rceil =\lceil \log_2 n\rceil+1$，或 $k=\lfloor \log_2 n+2\rfloor =\lfloor \log_2 n\rfloor+2$。

（3）设深度为 k，则叶子数 $n\leqslant 2^{k-1}$，得 $\log_2 n+1\leqslant k$，故 $k\geqslant\lceil \log_2 n+1\rceil =\lceil \log_2 n\rceil+1$。

5.10 解：已知叶子数的完全二叉树一般有两颗（见习题 5.9）。叶子数为 2^{k-1} 的完全二叉树可以是：（1）深度为 k 的满二叉树。（2）深度为 k+1，最底层只有 1 个结点，倒数第二层除了最左边的 1 个结点外，其他都是叶子，即叶子数仍为 $(2^{k-1}-1)+1=2^{k-1}$。

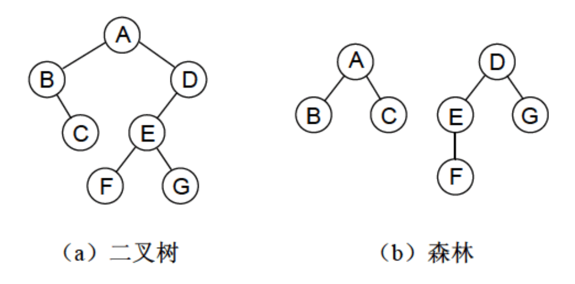
5.11 证：除根结点外，其他每个结点都是某个结点的孩子，即孩子结点数为 n−1。每个孩子对应其双亲的一个分支，于是所有孩子的总数就是所有结点的分支数之和，后者即所有结点的度数之和，所以 $n-1=\sum_{i=1}^{n}D(i)$，即 $n=\sum_{i=1}^{n}D(i)+1$。

5.12 证：由树到二叉树的转换，树中每个分支结点，在二叉树中有一个左孩子；该孩子结点的右孩子，右孩子的右孩子，……，在原树中是兄弟关系，其中最后一个没有右兄弟，它转换后没有右孩子。于是每个分支结点得到一个右子树为空的结点。

另外，森林各树的根之间是兄弟关系，最后一个根没有右兄弟，它转换后也没有右孩子。所以总共有 n+1 个结点没有右孩子。

5.13 解：层序序列的第一个结点 A 即为根，再从中序序列可分出其左、右子树，即 A 有两个子树。于是层序序列的第 2、3 结点 B、C 即为 A 的左、右子树的根，在中序序列中又可分出它们各自的左、右子树。依次类推，可逐步求出二叉树，其中层序序列的每一个结点都是当前子树的根。结果如右图所示。

5.14 解：森林的先根、后根序列分别与对应二叉树的先根和中根序列相同，于是可先画出二叉树，再转换成森林。结果如下图所示。



（a）二叉树　　（b）森林

5.15 解：线索二叉链表中仍然有空指针，但含义不同，线索为空是指结点没有相应遍历的前趋或后继，左、右指针为空是指结点没有左、右孩子。一般中序线索二叉链表中有两个空指针，先序和后序线索二叉链表中有一个或两个空指针。

5.16 解：

（1）二叉树的高度=左子树和右子树中高度较大者+1(根)。算法如下：

```
int high(bitree t) {              //返回二叉树的高度
  int L,R;
  if(t==NULL) return 0;           //当前树为空，递归出口
  L=high(t->lchild);              //求左子树高度
  R=high(t->rchild);              //求右子树高度
  return (L>R?L:R)+1;
}
```

（2）如果根结点的度不为 1，则二叉树中度为 1 的结点数=左子树中度为 1 的结点数+右子树中度为 1 的结点数；若根结点的度为 1，则要在此基础上再加 1（根），算法如下：

```
int sum1(bitree t) {              //返回二叉树中度 1 的结点数
  int L,R;
  if(t==NULL) return 0;           //当前树为空，递归出口
  L=sum1(t->lchild);              //求左子树中度为 1 的结点数
  R=sum1(t->rchild)               //求右子树中度为 1 的结点数
  if((t->lchild==NULL && t->rchild!=NULL) ||
     (t->lchild!=NULL && t->rchild==NULL))  //当前结点(根)的度也为 1
    return L+R+1;
  else
    return L+R;
}
```

类似可求度为 2 的结点数，同样要根据当前根的度是否为 2 来返回 L+R 或 L+R+1。

（3）按先根遍历思想进行查找，算法如下：

```
pointer search(bitree t,datatype x) {  //返回查找到的结点的地址
  pointer p;
  if(t==NULL) return NULL;        //空树
  if(t->data==x) return t;        //访问根：若找到就返回之(跳过子树的检查)
  p=search(t->lchild) || p=search(t->rchild);  //左子树没有，则找右子树
  return p;
}
```

5.17 解：

（1）在二叉树遍历的各种非递归算法上修改即可，即将访问根修改为交换其左、右孩子。以课本第 3 种非递归先根遍历算法为例，交换算法如下：

```
void exchange3(bitree t) { //交换各结点左、右子树，非递归先根遍历型算法，根预入栈
  pointer p,q,S[maxsize];  //顺序栈
  int top;                 //栈顶指针
  if(t==NULL) return;
  top=-1;
  S[++top]=t;              //根指针入栈
  while(top>=0) {          //栈非空，循环
    p=S[top--];            //出栈
    q=p->lchild;p->lchild=p->rchild;p->rchild=q;//交换
    if(p->lchild!=NULL) S[++top]=p->lchild;     //原右指针入栈
    if(p->rchild!=NULL) S[++top]=p->rchild;     //原左指针入栈
  }
}
```

（2）在二叉树遍历的各种非递归算法上修改即可，即将访问根修改为判断它是否为叶子，若是则叶子计数器加 1。具体算法略。

5.18 解：

（1）方法 1：在非递归后根遍历基础上修改：当后根遍历访问到结点*s 时，栈中的所有结点便是*s 的祖先，从而得到从根到该结点的路径，具体算法略。

方法 2：递归判断：若 p 在 t 的左子树或右子树中，则 t 为 p 的祖先，输出之，算法如下：

```
int ancestor(bitree t,pointer p) {
  if(t==NULL) return 0;
  if(t==p) return 1;
  if(ancestor(t->lchild,p) || ancestor(t->rchild,p)) {
    cout<<t->data<<endl;
    return 1;
  }
  return 0;
}
```

注意，该算法输出的是 p 的双亲到根的路径。

（2）方法 1：类似上题方法 1，用非递归后根遍历，分别求出*p 和*q 的祖先，从根开始对比，即可求出它们的共同祖先。具体算法略。

方法 2：类似上题方法 2，递归判断：若 p、q 在 t 的左子树或右子树中，则 t 为 p、q 的共同祖先，输出之。具体算法略。

5.19 解：

（1）注意二叉树的顺序存储结构中可能有多个虚结点。算法如下：

```
int leaf(int A[],int n) {
  int i,num;
  num=0;
  for(i=1;i<=n;i++) {
    if(A[i]==0) continue;    //虚结点
    if(2*i>n) num++;         //孩子编号超出范围，必为叶子
    else if(A[2*i]==0 && A[2*i+1]==0) num++;//叶子
  }
  return num;
}
```

（2）顺序存储结构对应于层序序列，故可按层序序列建立二叉链表，算法见课本（需略作修改）。由于层序序列已存于数组中，还可利用双亲与孩子结点的编号关系进行建立，算法如下：

```
bitree creat(int i) {                //i 为结点序号（设 A[n]和 n 为全局量）
  pointer t;
  if(i>n || A[i]==0) return NULL;    //该结点不存在
  t=new node;
  t->data=A[i];
  t->lchild=creat(2*i);
  t->rchild=creat(2*i+1);
  return t;
}
```

（3）这里不能直接按层序序列存储数组 A，因为层序遍历中不输出空（虚）结点。下面的算法思想是将输出的结点直接存到对应的数组位置。为此，对某个结点，需要知道它的序号 i，这可利用双亲与孩子结点的编号关系，从根（i=1）开始，沿左、右子树 2*i、2*i+1

逐个计算出来。算法如下：

```
void creat(bitree t,int i) {        //i 为结点序号（设 A[n]和 n 为全局量）
  pointer t;
  if(i>n) return;                   //该结点不存在
  if(t==NULL) {A[i]=0);return;}     //该结点不存在
  A[i]=t->data;
  creat(t->lchild,2*i);
  creat(t->rchild,2*i+1);
}
```

5.20 解：

（1）按安全二叉树的层序编号规则将结点存于一维数组，若其中无虚结点就是完全二叉树。但这里不需将结点全部存完再检测虚结点，而是边存储边检测。算法如下：

```
int detect(bitree t) {        //按层序编号判断完全二叉树
  pointer A[maxsize+1];       //maxsize 足够大，如取为同高度的满二叉树结点数
  int i,n;
  if(t==NULL) return 1;
  for(i=1;i<=maxsize;i++) A[i]=NULL;
  i=n=1;                      //n 为当前结点编号最大值
  A[1]=t;                     //1 号位置为根结点
  while(i<=n) {
    if(A[i]==NULL) return 0;//虚结点
    if(A[i]->lchild!=NULL)
      {n=2*i;  A[n]=A[i]->lchild;}   //对左孩子编号并存储
    if(A[i]->rchild!=NULL)
      {n=2*i+1;A[n]=A[i]->rchild;}   //对右孩子编号并存储
    i++;
  }
  return 1;
}
```

（2）所谓二叉树的宽度是指某一层上最多的结点数。由层序遍历的特点可知，对某层第一个（最左边）结点遍历时对应其孩子层的开始；对某层最后一个（最右边）结点遍历后对应其孩子层的结束。据此从根开始可依次求出以下各层的宽度。

层序遍历需采用队列结构。设 front 指向当前其孩子正在遍历的结点（假设为第 i 层），rear 指向当前正在遍历的结点（为 front 的孩子，为第 i+1 层），b 指向第 i 层最右边的结点（为其上一层的最后一个孩子，即上一层最后的 rear），则当 front=b 时第 i 层所有点的孩子都遍历完，于是可求出孩子层的宽度。算法如下：

```
int width(bitree t) {         //层序遍历求宽度
  pointer p,Q[maxsize];       //队列
  int front,rear,b;
  int w,count;                //w 记录各层最大宽度，count 累计当前层的宽度
  if(t==NULL) return 0;
  front=rear=-1;
  Q[++rear]=t;                //根结点入队
  w=1;                        //根层宽度为 1
  count=0;
  b=rear;
  while(front<b) {            //当前层未处理完
    front++;p=Q[front];       //出队，访问队头
```

```
      if(p->lchild!=NULL) {
        Q[++rear]=p->lchild;count++;//左孩子入队，累计孩子层宽度
      }
      if(p->rchild!=NULL) {
        Q[++rear]=p->rchild;count++;//右孩子入队，累计孩子层宽度
      }
      if(front==b) {            //当前层处理完
        if(w<count) w=count;
        count=0;
        b=rear;                 //b 指向下一层最右边的结点
      }
    }
    return w;
  }
```

注意，while 循环的条件不是队列非空，因为最后一层不需要遍历，它没有孩子层。

（3）可对上题算法修改，每一层结点的层数都相同；某一层遍历完后层数加 1；最后一层的层数即树的高度，算法略。

另外，也可用一个数组 level[maxsize]记录每个已访问结点的层数，则下次访问其孩子时，孩子层数为该点层数+1，这样可不必判断某层的开始和结束，算法如下：

```
int high(bitree t) {          //层序遍历求高度
  pointer p,Q[maxsize];       //队列
  int front,rear;
  int level[maxsize],hmax,h;
  if(t==NULL) return 0;
  hmax=0;front=rear=-1;
  rear++;Q[rear]=t;Level[rear]=1;       //根结点入队,根层数为 1
  while(front!=rear) {                  //队列非空，循环
    front++;p=Q[front];h=level[front]; //出队，访问队头
    if(h>hmax) hmax=h;
    if(p->lchild!=NULL) {
      rear++;Q[rear]=p->lchild;level[rear]=h+1;
    }
    if(p->rchild!=NULL) {
      rear++;Q[rear]=p->rchild;level[rear]=h+1;
    }
  }
  return hmax;
}
```

5.21 解：

（1）注意到在森林（或树）对应的二叉树中，若某结点没有左孩子（长子），则其为叶子；以及结点与其左子树对应一棵树，结点的右子树对应其他树组成的森林，则算法如下：

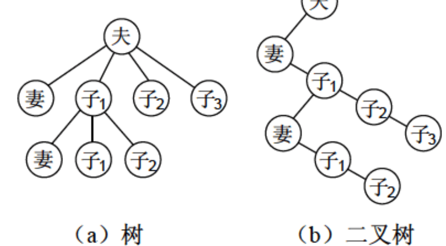
```
int leaf(bitree t) {          //求森林或树的叶子数
  if(t==NULL) return 0;
  if(t->lchild==NULL) return 1+leaf(t->rchild);
  return leaf(t->lchild)+leaf(t->rchild);
}
int high(bitree t) {          //求森林或树的高度
  int L,R;
  if(t==NULL) return 0;
  L=1+high(t->lchild);        //第一棵树的高度
  R=high(t->rchild);          //其他树组成的森林的高度
```

```
  return L>R?L:R;
}
```

(2) 在森林（或树）对应的二叉树中，结点与其右孩子，右孩子的右孩子，……，为同层结点，对它们可依次访问。下一层的结点依次从上层结点的左孩子开始，同样沿右孩子搜索。整个过程相当于对二叉树按斜右下的方向层序遍历。对每个结点，需用队列记住其左孩子以便下一层遍历。算法如下：

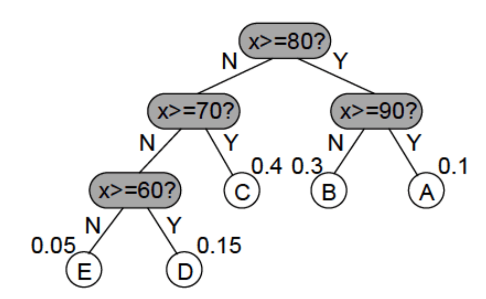
```
void level(bitree t) {        //层序遍历森林或树的二叉树
  sqqueue Q;                  //队列，类型为 pointer
  pointer p;
  if(t==NULL) return;
  init_sqqueue(&Q);
  en_sqqueue(&Q,t);           //根结点入队
  while(!empty_sqqueue(&Q)) {
    de_sqqueue(&Q,&p)         //出队
    while(p!=NULL) {
      if(p->lchild!=NULL) en_sqqueue(&Q,p->lchild); //左孩子入队
      cout<<p->data<<" ";
      p=p->rchild;            //向右搜索
    }
  }
}
```

5.22 解：对任一棵树，表示父子和兄弟关系是显然的。如果还要表示夫妻关系，可对树施加一定的约束，如将原孩子中长子（或幼子）的位置表示夫妻，其他位置表示孩子，见右图(a)；与该图相应的二叉树便可表示父子、兄弟、夫妻三种关系了，见右图(b)。



(a) 树　　(b) 二叉树

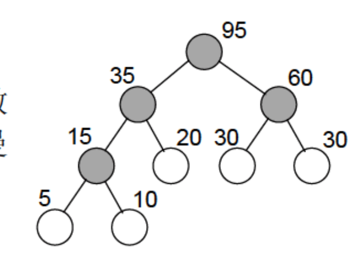
5.23 解：判定树见右图，相应的算法如下：




```
char trans(float x) {
  if(x>=80)
    if(x>=90) return 'A';
    else      return 'B';
  else if(x>=70) return 'C';
  else if(x>=60) return 'D';
  else           return 'E';
}
```

平均比较次数 n=0.05×3+0.15×3+0.4×2+0.3×2+0.1×2=2.2

5.24 解：文件每合并一次，其中的每个记录都要移动一次；即合并多少次，有关的记录就要移动多少次。于是取合并的次数为路径长度，每个文件作叶子，取其中的记录数作权构造哈夫曼树即可，结果如右图所示。



第 6 章　图

6.1 解：(1) 顶点度最大为 n−1（无向图）或 2(n−1)（有向图，每个顶点到其余 n−1 个顶点都可有一条入边和出边）。由于 $\sum D(v_i) = 2e$，为偶数，所以度为奇数的点只能是偶数个（否则奇数个奇数的和仍为奇数）。

（2）设顶点数为 n、边数为 e。由已知得所有顶点度数之和 $\Sigma \geqslant 2n$；由于每条边有两个顶点，又可得所有顶点度数之和 $\Sigma=2e$，于是有 $e \geqslant n$。但 n 个顶点若无回路，则最多 $n-1$ 条边，即生成树，超过 $n-1$ 条边后肯定出现回路。

6.2 解：（1）由 $e \leqslant n(n-1)/2$，得 $25 \times 2 \leqslant n(n-1)$，解此不等式，得 $n \geqslant 7.59$，即 n 至少为 8。或者简单试凑 $50 \nleqslant 7(7-1)=42$、$50 \leqslant 8(8-1)=56$ 得 n 至少为 8。

（2）无边时 10 个顶点各自孤立，则有 10 个连通分量，有边时会引起连通分量的合并而减少。

① 连通分量数最多时，由 5 条边合并的连通分量数要最少。这时所有边在同一个连通分量，且其中顶点数尽可能少，也即尽量接近完全图。对 5 条边，由 $e \leqslant n(n-1)/2$，得连接的顶点数最少为 4 个，还剩 6 个顶点为孤立点，故最多有 1+6=7 个连通分量。

② 连通分量数最少时，由 5 条边合并的连通分量数要最多。这时每条边使两个连通分量合二为一，即每条边减少一个连通分量，故剩余连通分量数=10−5=5 个。

6.3 解：（1）设有 m 棵树，每棵树的顶点有 v_i 个，则其边数为 v_i-1，于是：

$$v_1+v_2+\cdots+v_m=n$$

$$(v_1-1)+(v_2-1)+\cdots+(v_m-1)=e$$

两式相减可得 $m=n-e$。

（2）注意每个连通分量对应一棵树，利用上题结果有 $e=n-m$。

6.4 解：（1）邻接矩阵为上三角矩阵，则任一顶点的邻点号都比该点的大，故从任一顶点出发的路径都不会回到该顶点自己，即图中没有回路，故可以拓扑排序。

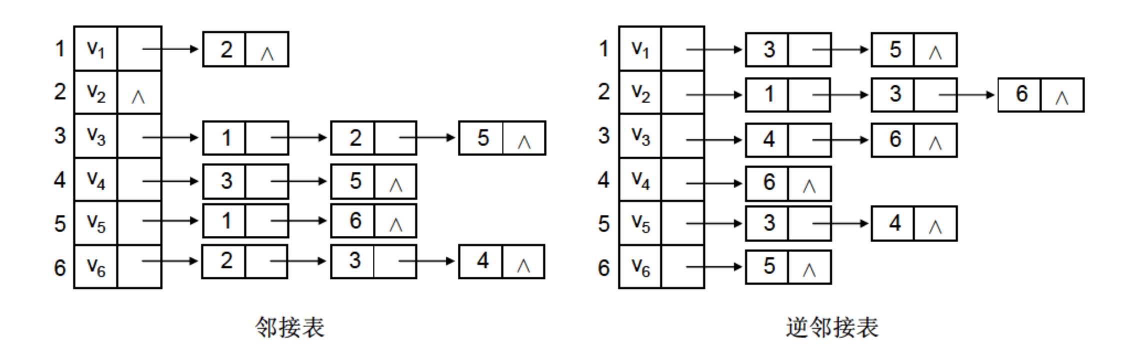
（2）沿拓扑序列对顶点重新编号，则任一顶点的邻点号都比该点的大，故邻接矩阵为上三角矩阵（但不一定全是 1，不存在的边对应 0）。

6.5 解，方法 1：对该图进行拓扑排序，若能完成则无回路；否则有回路。

方法 2：对该图进行 DFS 遍历，如果从某点出发的搜索过程中又回到了该出发点（即遇到了一条指向出发点的回边），则有回路。或者说，如果从任意点出发的搜索过程中都不会回到出发点，则没有回路。

6.6 解：邻接表上求出度方便，但求入度不方便，从而求度也不方便；邻接矩阵上求出度和入度都方便（看相应的行、列上的非零元）。所以用邻接矩阵较好（其他几个运算两者都方便）。

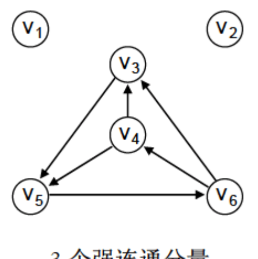
6.7 解：结果见下图所示。



邻接表　　　　逆邻接表

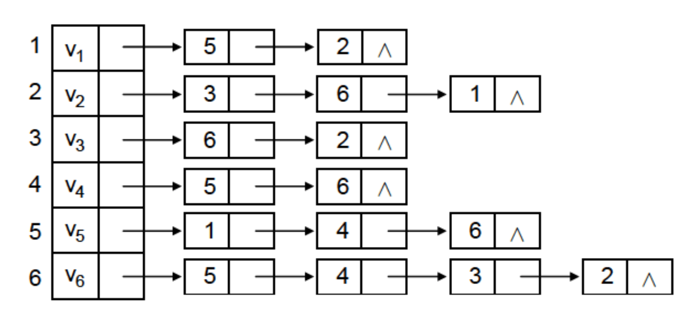
$$A=\begin{bmatrix}0&1&0&0&0&0\\0&0&0&0&0&0\\1&1&0&0&1&0\\0&0&1&0&1&0\\1&0&0&0&0&1\\0&1&1&1&0&0\end{bmatrix}$$

邻接矩阵



3 个强连通分量

6.8 解：由于是无向图，每输入 1 条边要在邻接表上建立 2 个结点，用头插法建立的邻接表如下图所示：



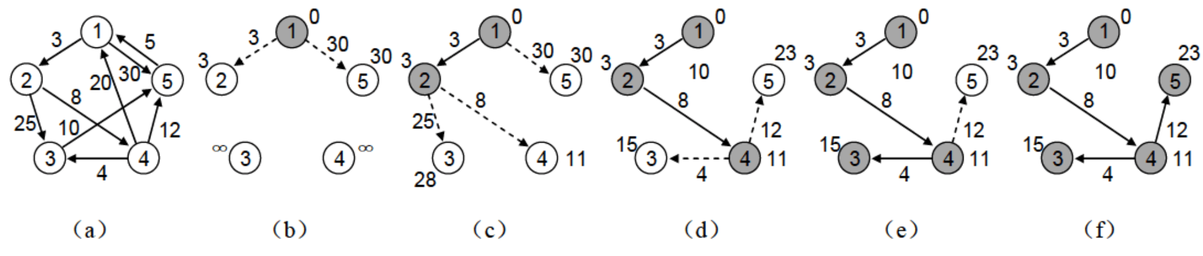
在此邻接表上从顶点 v_2 出发的 DFS 遍历序列为 $v_2, v_3, v_6, v_5, v_1, v_4$，BFS 遍历序列为 $v_2, v_3, v_6, v_1, v_5, v_4$。

6.9* 解：步骤如下：

（1）从任一顶点出发按出边方向进行 DFS 遍历，并按其所有邻接点的遍历都完成的顺序将顶点排列起来。

（2）从最后完成遍历的顶点出发按入边方向进行逆向 DFS 遍历，若不能访问到所有顶点，则从余下的顶点中最后完成遍历的顶点出发继续作逆向 DFS 遍历，依次类推，直到所有顶点都访问到为止。其中每一次逆向 DFS 遍历所访问到的顶点集便是有向图的一个强连通分量。

6.10* 解：求解过程见下图：



（a）　（b）　（c）　（d）　（e）　（f）

6.11* 解：迭代过程和结果见下图：

$$A_0=\begin{bmatrix}0&1&\infty&4\\\infty&0&9&2\\3&5&0&8\\\infty&\infty&6&0\end{bmatrix}\qquad path_0=\begin{bmatrix}1&2&3&4\\1&2&3&4\\1&2&3&4\\1&2&3&4\end{bmatrix}$$

$$A_1=\begin{bmatrix}0&1&\infty&4\\ \infty&0&9&2\\ 3&4&0&7\\ \infty&\infty&6&0\end{bmatrix}\quad path_1=\begin{bmatrix}1&2&3&4\\1&2&3&4\\1&1&3&1\\1&2&3&4\end{bmatrix}$$

0：1->1
1：1->2
9：1->2->4->3
3：1->2->4

$$A_2=\begin{bmatrix}0&1&10&3\\ \infty&0&9&2\\ 3&4&0&6\\ \infty&\infty&6&0\end{bmatrix}\quad path_2=\begin{bmatrix}1&2&2&2\\1&2&3&4\\1&1&3&1\\1&2&3&4\end{bmatrix}$$

11：2->4->3->1
0：2->2
8：2->4->3
2：2->4

$$A_3=\begin{bmatrix}0&1&10&3\\ 12&0&9&2\\ 3&4&0&6\\ 9&10&6&0\end{bmatrix}\quad path_3=\begin{bmatrix}1&2&2&2\\3&2&3&4\\1&1&3&1\\3&3&3&4\end{bmatrix}$$

3：3->1
4：3->1->2
0：3->3
6：3->1->2->4

$$A_4=\begin{bmatrix}0&1&9&3\\ 11&0&8&2\\ 3&4&0&6\\ 9&10&6&0\end{bmatrix}\quad path_4=\begin{bmatrix}1&2&2&2\\4&2&4&4\\1&1&3&1\\3&3&3&4\end{bmatrix}$$

9：4->3->1
10：4->3->1->2
6：4->3
0：4->4

6.12* 解：

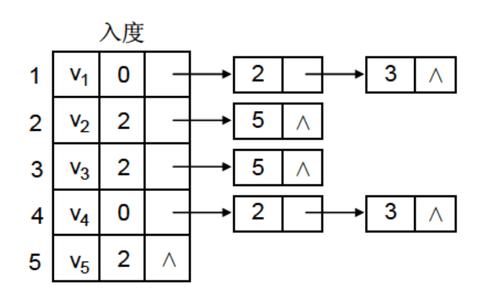
（1）其他村庄到某村庄的最小距离对应距离矩阵的各列。由 A_4 可见，取 4 号村庄建俱乐部较好，其他村庄到它的距离为 3, 2, 6（矩阵第 4 列），距离分配既均匀又都较短。

（2）某村庄到其他村庄的最小距离对应距离矩阵的各行。由 A_4 可见，取 3 号村庄建俱乐部较好，它到其他村庄的距离为 3, 4, 6（矩阵第 3 行），距离分配既均匀又都较短。

（3）考察距离矩阵中各列和的最小值，分别为 23, 15, 23, 11，故取 4 号村庄较好。

（4）考察距离矩阵中各行和的最小值，分别为 13，21，13，25，故取 1 号或 3 号村庄较好。

6.13 解：邻接表见下图：



用栈保存入度为 0 的点，则有多个入度为 0 的点时输出的是后保存的点，所以得到拓扑排序序列为：v_4, v_1, v_3, v_2, v_5。

用队列保存入度为 0 的点，则有多个入度为 0 的点时输出的是先保存的点，所以得到拓扑排序序列为：v_1, v_4, v_2, v_3, v_5。

6.14 解：

（1）各事件的最早发生时间：

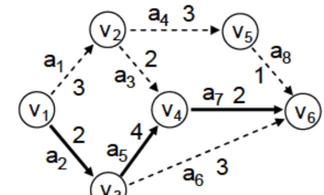
```
ve(1)=0
ve(2)=ve(1)+w12=0+3=3
ve(3)=ve(1)+w13=0+2=2
ve(4)=max{ve(2)+w24,ve(3)+w34}=max{3+2,2+4}=6
ve(5)=ve(2)+w25=3+3=6
ve(6)=max{ve(5)+w56,ve(4)+w46,ve(3)+w36}=max{6+1,6+2,2+3}=8
```

（2）各事件的最晚发生时间：

```
vl(6)=ve(6)=8
vl(5)=vl(6)-w56=8-1=7
vl(4)=vl(6)-w46=8-2=6
vl(3)=min{vl(4)-w34,vl(6)-w36}=min{6-4,8-3}=2
vl(2)=min{vl(5)-w25,vl(4)-w24}=min{7-3,6-2}=4
vl(1)=min{vl(2)-w12,vl(3)-w13}=min{4-1,2-2}=0
```

（3）各活动的最早开始时间 e、最迟开始时间 l，以及时间余量如下表所示，关键路径如下图实线所示。

活动	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
e	0	0	3	3	2	2	6	6
l	1	0	4	4	2	5	6	7
l–e	1	0	1	1	0	3	0	1



6.15 解：先建立顶点表，然后依次读入边<i, j>，对每条边生成 1 个边表结点，其 no 域为 i，将它插入到第 j 个边表中。采用头插法。在顶点对输入过程中，自动累计边数，若输入的顶点号 i<0 则结束。算法如下：

```
void creatgraph(lk_graph *ga) {
  int i,j,n,e;
  pointer p;
  cin>>n;                //读入顶点数
  ga->n=n;
  for(i=1;i<=n;i++) {    //读入顶点信息，建立顶点表
    cin>>ga->adjlist[i].data;
    ga->adjlist[i].first=NULL;
  }
  e=0;
  while(cin>>i>>j,i>0) {//读入边，建立边表
    e++;                 //累计边数
    p=new node;          //生成邻接点序号为 i 的表结点
    p->no=i;
    p->next=ga->adjlist[j].first;
    ga->adjlist[j].first=p;//将新表结点插入到顶点 vj 的边表头部
  }
  ga->e=e;
}
```

6.16 解：由于要求邻接表中的结点按顶点序号的大小顺序排列，所以在邻接矩阵某行上求邻接点时，如果是从左向右扫描，则要用尾插法建邻接表；如果是从右向左扫描，则要用头插法建邻接表。以后者为例，算法如下：

```
void mattolist(mat_graph *gm,lk_graph *gl) {
  int i,j,n,e;
  pointer p;
```

```
    gl->n=gm->n;gl->e=gm->e;
    for(i=1;i<=gl->n;i++) gl->adjlist[i].first=NULL;
    for(i=1;i<=gm->n;i++)
      for(j=gm->n;j>=1;j--) {
        if(gm->adjmat[i][j]==0) continue;
        p=new node;                  //生成邻点
        p->no=j;                     //插入到表头
        p->next=gl->adjlist[i].first;
        gl->adjlist[i].first=p;
      }
}
```

6.17 解：这里需要将访问过的顶点入栈。假设图以邻接表表示，算法如下：

```
void dfsL(lk_graph *g,int v) { //邻接表 DFS 遍历,非递归
  int S[nmax],top;             //栈
  pointer p;
  top=-1;
  cout<<v<<" ";visited[v]=1;   //访问出发点，假设为输出顶点序号
  S[++top]=v;
  while(top!=-1) {
    p=g->adjlist[S[top]].first;
    while(p!=NULL && visited[p->no]) p=p->next;//搜索栈顶未访问的一个邻接点
    if(p==NULL) top--;                          //退到前一个顶点
    else {
      cout<<p->no<<" ";visited[p->no]=1;        //访问出发点
      S[++top]=p->no;
    }
  }
}
```

6.18 解：

（1）假设有向图以邻接表表示。从 v_i 出发进行非递归 DFS 遍历（将访问过的顶点入栈），若遍历中遇到顶点 v_j，则顶点 v_i 到 v_j 有路径，此时栈内的顶点序列即为路径。算法如下：

```
int pathdetect(lk_graph *g,int i,int j) {
  int S[nmax],top;          //栈
  int k;
  pointer p;
  top=-1;
  visited[i]=1;
  S[++top]=i;
  while(top!=-1) {
    p=g->adjlist[S[top]].first;
    while(p!=NULL && visited[p->no]) p=p->next;//搜索栈顶未访问的一个邻接点
    if(p==NULL) top--;
    else {
      visited[p->no]=1;
      S[++top]=p->no;
      if(p->no==j) {         //搜索到 vj
        cout<<"发现路径：\n";
        for(k=0;k<=top;k++) cout<<S[k]<<" ";
        return 1;
      }
    }
  }
  cout<<"没有路径：\n";
  return 0;                  //始终没有搜索到 vj
}
```

（2）可在（1）的基础上修改，即每搜索到一个结点，就检测路径上的结点总数（即 top）是否等于图的结点总数 g->n。这里没有指定出发点，可依次从每个结点出发进行上述算法，具体算法略。

6.19 解：

（1）本题与上题（1）类似，但这里访问出发点 v 后不能将其访问标志置 1，否则以后搜索未访问的邻接点时不可能再找到 v。算法如下：

```
int pathdetect3(lk_graph *g,int v) {
  int S[nmax],top;          //栈
  int k;
  pointer p;
  top=-1;
  S[++top]=v;
  while(top!=-1) {
    p=g->adjlist[S[top]].first;
    while(p!=NULL && visited[p->no]) p=p->next;//搜索栈顶未访问的一个邻接点
    if(p==NULL) top--;
    else {
      visited[p->no]=1;
      S[++top]=p->no;
      if(p->no==v) {        //搜索到 v
        cout<<"发现回路：\n";
        for(k=0;k<=top;k++) cout<<S[k]<<"  ";
        return 1;
      }
    }
  }
  cout<<"没有回路：\n";
  return 0;                 //始终没有搜索到 vj
}
```

（2）可在（1）的基础上修改，即当出现回路时，检测路径上的结点总数（即 top）是否等于图的结点总数 g->n。这里没有指定出发点，可依次从每个结点出发进行上述算法，具体算法略。

6.20 解：

（1）采用 BFS 遍历，距离顶点 v 层数为 len+1 的顶点就是最短路径长度为 len 的顶点。遍历中保存各点的层数，以便求其邻接点的层数（为该点层数+1）。以邻接表为例，算法如下：

```
void search(lk_graph *g,int v,int len) {
  sqqueue Q1,Q2;              //假设采用顺序队列，分别保存访问的顶点号及其层数
  pointer p;
  int level;
  init_sqqueue(&Q1);init_sqqueue(&Q2);
  visited[v]=1;level=1;
  en_sqqueue(&Q1,v);en_sqqueue(&Q2,level);
  while(!empty_sqqueue(&Q1) && level<len+1) {
    de_sqqueue(&Q1,&v);de_sqqueue(&Q2,&level);
    p=g->adjlist[v].first;
```

```
            level++;
            while(p!=NULL) {
              if(!visited[p->no]) {
                if(level==len+1) cout<<p->no<<"  ";
                visited[p->no]=1;
                en_sqqueue(&Q1,p->no);en_sqqueue(&Q2,level);
              }
              p=p->next;
            }
          }
        }
```

（2）可在（1）的基础上修改，即这些顶点就是最后一层上的所有结点。算法略。

第 7 章　排序

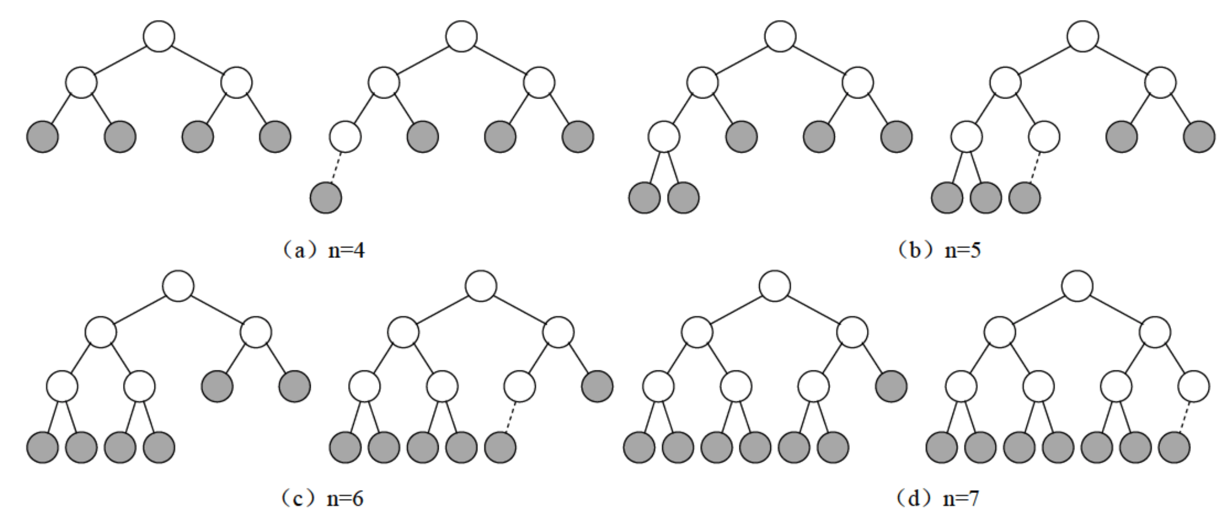
7.1 略

7.2 略

7.3 略

7.4 略

7.5 解：已知叶子数的完全二叉树一般有两颗（见习题 5.9），所以树形选择排序也可有两种可能，其中最底层的叶子数分别为偶数和奇数，但后一种在两两比较时有一个叶子轮空直接进入上一层，故实际只需考虑前一种情况。具体结果见如下示意图。



（a）n=4　　（b）n=5

（c）n=6　　（d）n=7

7.6 解：在 n 个数据中找最小的，任何基于比较的排序方法至少需要作 n−1 次比较。为使找次小时比较次数最少，就要充分利用前一次比较的结果，这可用二叉树将前一次比较的结果记录下来。故采用树形选择排序。它在选出最小后，以后各趟只需比较 $\lceil \log_2 n \rceil$ 次。所以本问题总的比较次数为 $n-1+\lceil \log_2 n \rceil=1\,009$。但这个结果中有一次是与+∞比较，是不必要的，所以上述结果应减 1。另外，1 000 个结点的完全二叉树不是满二叉树，最小叶子可能不在最底层而在倒数第二层，这时上述结果应再减 1。所以最少比较次数为 1009−1 或 1009−2（取决于数据分布情况）。

7.7 解：小根堆中最大者只可能是叶子，1 000 个结点的堆有 500 个叶子，找其中最大，至少比较 500−1 次。

7.8 解：

方法 1：用快速排序，每趟划分后检查基准位置 i（它就是其最终位置），若 i=s 或 s+1，则基准左侧即为所求；若 i>s+1 或 i<s，再对左序列或右序列继续划分。

方法 2：用堆排序，建立初始小根堆后，进行 s−1 次重建和调整。

方法 3：用冒泡排序（上升法），进行 s 趟，得到 s 个最小。

方法 4：用直接选择排序，进行 s 趟，选出 s 个最小。

7.9 解：以课本的划分算法 2 为例：从右向左扫描，若有交换则改变方向；从左向右扫描，若有交换再改变方向，……。

（1）每个关键字都移动一次，则关键字中后面的都比基准小，前面的都比基准大。

对 7 个关键字，除基准外还有 6 个，故比基准小和比基准大的各 3 个，比如 4, 5, 6, 7, 1, 2, 3，移动次数=6+2=8，其中基准移动 2 次：划分开始移走、结束时移回。

对 8 个关键字，除基准外还有 7 个，由于先从右向左扫描，故比基准小的有 4 个，比基准大的有 3 个，比如 5, 6, 7, 8, 1, 2, 3, 4，移动次数=7+2=9。

（2）移动次数最少，需划分趟数最少，且每趟的移动最少。前者要求每趟得到两个等长区间，即基准位于中间位置，则该位置原来的元素必定和基准交换一次；后者要求其他元素不发生交换。所以中间位置后的元素比基准大，前面的比基准小；划分后的区间有同样要求。比如序列 4, 1, 3, 2, 6, 5, 7。共划分 2 趟，出现 3 个基准，与此对应有 3 次元素移动，而每个基准移动 2 次，故总的移动次数为 3+3×2=9。

（3）比较次数最少即最好情况，每趟划分得两个等长区间。当 n=7 时，第一趟划分时比较 6 次，得两个区间，长度各为 3。第二趟划分时，两个区间各比较 2 次，分别得两个长度为 1 的区间，排序结束。总共比较了 6+2×2=10 次，比如序列 4, 7, 5, 6, 3, 1, 2。

（4）比较次数最多即最坏情况，每趟划分后一个区间长度为 0，另一个长度为原区间长度减 1。当原序列有序时，出现这种情况。当 n=7 时，总比较次数为 6+5+4+3+2+1=21，比如序列 7，6, 5, 4, 3, 2, 1 或 1, 2, 3, 4, 5, 6, 7。

（5）任何基于比较的排序，最坏情况下关键字的比较最少 $\lceil \log_2(n!) \rceil$ 次。

注意，这个下界可能是达不到的，如 n=12 时，$\lceil \log_2(12!) \rceil=29$，有人用穷举法在计算机上证明不可能用 29 次比较来完成 12 个数的排序，实际最优方案至少需要 30 次比较。

7.10* 解：对于快速排序的平均性能，要考虑每次划分后基准出现在各种位置上的情况。假设基准在每个位置上出现的概率是相同的。若基准的位置为 k（k=1, 2, …, n），则划分后两个区间长度分别为 k−1 和 n−k，于是平均比较次数为：

$$C(n) = n-1+\frac{1}{n}\sum_{k=1}^{n}[C(k-1)+C(n-k)] \quad (n \geqslant 1)，其中\ C_0=0, C_1=0, C_2=1, \cdots$$

其中 n−1 为 n 个数据划分时的比较次数。注意到 $\sum_{k=1}^{n}C(k-1)$ 和 $\sum_{k=1}^{n}C(n-k)$ 实际上是相等的，因为一个从 C(0)累加到 C(n−1)，另一个从 C(n−1)累加到 C(0)，所以：

$$C(n) = n-1+\frac{2}{n}\sum_{k=0}^{n-1}C(k) \quad (n \geqslant 1)$$

其通式为（见附录 E）：

$$C(n) = 2(n+1)\ln(n+1) + O(n)$$
$$\approx 2n\ln(n) \approx 1.39n\log_2 n = O(n\log_2 n)$$

记录移动次数不大于比较的次数，故快速排序的平均时间复杂度为 $O(n\log_2 n)$。

7.11 解：算法如下：

```
void InsertSort2(list R,int n) {
  int i,j,low,high,mid;
  for(i=2;i<=n;i++) {            //依次插入 R[2],R[3],…,R[n]
    if(R[i].key>=R[i-1].key) continue;//R[i]大于有序区最后一个记录，不需插入
    R[0]=R[i];
    low=1;high=i-1;
    while(low<=high) {           //查找 R[i]的插入位置
      mid=(low+high)/2;
      if(R[0].key<R[mid].key) high=mid-1;
      else                     low=mid+1;
    }
    for(j=i-1;j>=mid;j--)
      R[j+1]=R[j];               //记录后移
    R[mid]=R[0];                 //插入 R[i]
  }
}
```

7.12 解：采用监视哨时，每组都需一个监视哨。设当前增量为 h，则数据表分为 h 组：$(R_i, R_{i+h}, R_{i+2h}, \cdots)$，i=1, 2, …, h，由于组内记录间下标相差 h，则各组监视哨的位置分别为 1−h, 2−h, …, 0。这些位置越出了数组下标范围，需要移动数据表，处理起来不方便。

为此，对算法作个变形，将分组和排序都从右向左进行，这时分组情况为：$(\cdots, R_{n-i-2h}, R_{n-i-h}, R_{n-i})$，i=0, 1, …, h−1，则每组监视哨的位置为 n−i+h，它们处于数据表的右端，这只要在数组上端预留一定空间即可。具体算法如下：

```
void ShellInsert(list R,int n,int h) {//一趟插入排序，从右向左进行，h 为本趟增量
  int i,j,k;
  for(i=0;i<h;i++) {             //i 为组号
    for(j=n-i-h;j>=1;j-=h) {     //每组从右第 2 个记录开始插入，它的下标为 n-i-h
      R[n-i+h]=R[j];             //监视哨
      k=j+h;                     //待插记录的后一个记录
      while(R[n-i+h].key>R[k].key) {//查找正确的插入位置
        R[k-h]=R[k];             //前移记录
        k=k+h;                   //向后搜索
      }
      R[k-h]=R[n-i+h];           //插入 R[j]
    }
  }
}
```

7.13 解：双向扫描时区间的两个端点都在变化，且任一个扫描方向若没有发生交换就可结束。算法如下：

```
void BubbleSort2(list R,int n) {//双向冒泡排序
  int i,j,k,flag;
  i=1;j=n;                       //扫描区间
  while(i<j) {
    flag=0;                      //置未交换标志
```

```
    for(k=j;k>=i+1;k--)          //从下向上扫描
      if(R[k].key<R[k-1].key) {  //交换记录
        flag=1;
        R[0]=R[k];R[k]=R[k-1];R[k-1]=R[0];  //交换，R[0]作辅助量
      }
    if(!flag) break;             //本趟未交换过记录，排序结束
    i++;                         //扫描区间缩短
    flag=0;                      //置未交换标志
    for(k=i;k<=j-1;k++)          //从上向下扫描
      if(R[k].key>R[k+1].key) {  //交换记录
        flag=1;
        R[0]=R[k];R[k]=R[k+1];R[k+1]=R[0];  //交换，R[0]作辅助量
      }
    if(!flag) break;             //本趟未交换过记录，排序结束
    j--;                         //扫描区间缩短
  }
}
```

7.14 解：

（1）由于是单链表，在有序区中寻找插入位置时只能从前向后进行。设链表有头结点，有序区尾结点位置为 rear，算法如下。

```
void InsertSort2(lklist head) {
  pointer s,t,p,rear;
  if(head->next==NULL) return;  //链表为空
  rear=head->next;              //第一点为初始有序区
  while(rear->next!=NULL) {     //无序区非空，循环
    p=rear->next;               //p 为无序区第一点
    s=head;
    t=s->next;
    while(t->data<=p->data && t!=p) {
                                //从有序区第一点开始，寻找插入位置 t，s 为其前趋
      s=t;t=t->next;
    }
    if(t==p) rear=p;            //位置无需移动
    else {rear->next=p->next;s->next=p;p->next=t;} //插入
  }
}
```

易见，该算法是稳定的。但若把内层循环条件中的“<=”改为“<”，则是不稳定的。

（2）设链表有头结点，有序区尾结点位置为 rear。

```
void SelectSort2(lklist head) {
  pointer s,t,q,p,rear;
  rear=head;                    //初始有序区为空
  while(rear->next!=NULL) {     //无序区非空，循环
    q=rear;p=q->next;           //p 为最小值点，初值为无序区第一点(q 为其前趋)
    s=p;t=s->next;              //t 为搜索点，从无序区第二点开始(s 为其前趋)
    while(t!=NULL) {            //寻找最小点
      if(t->data<p->data) {q=s;p=t;}
      s=t;t=t->next;
    }
    q->next=p->next;            //最小点插入到有序区尾部
    p->next=rear->next
    rear->next=p;
```

```
      rear=p;
    }
  }
```

易见，该算法是稳定的。注意算法不是将找到的无序区最小点与无序区的第一点交换（就像顺序存储时那样，其结果是不稳定的），因为链表上物理交换不如改指针方便。

（3）从前向后扫描并比较相邻两结点，在位置不对而进行交换时修改指针关系，而不真正物理交换，具体算法略。

7.15 解：这是一个三路划分问题，可在快速排序的二路划分基础上改。以单向扫描为例，设扫描过的数据分成负值区、零值区和正值区，其中零值区首末位置为 s、t，当前扫描位置为 i。预先取出 R[1]，空出位置 t。若 R[i]比零大，已在正值区，继续；若等于零，则存到原 t 处，并把原 R[t+1]移到 i 处，空出 t+1 位置，零值区增大一位；若为负数，则把原 R[s]移到 t 处，把 R[i]存到 s 处，原 R[t+1]移到 i 处，空出 t+1 位置，零值区整体后移一位。最后把原 R[1]放到适当位置。显然复杂性为 O(n)。算法如下：

```
int way3(list R,int n) {      //三路划分,单向扫描，返回零值区首位置
  int i,s,t;
  rectype x;                  //辅助量(可用 R[0]代替)
  if(n<=1) return;
  s=t=1;                      //s,t 等值区始末端
  x=R[1];                     //始终空出位置 t,最初即 R[1]
  for(i=2;i<=n;i++) {
    if(R[i].key>0) continue;
    else if(R[i].key==0) {R[t]=R[i];R[i]=R[t+1];t++;}
    else   {R[t]=R[s];R[s]=R[i];R[i]=R[t+1];t++;s++;}
  }
  if(x.key>=0) R[t]=x;
  else    {R[t]=R[s];R[s]=x;s++;}
  return s;
}
```

采用双向扫描时，可在划分过程中，左侧等于零的数先放到区间左端；右侧等于零的数先放到区间右端，划分完后再将前后两端的零值区交换到区间中间位置。具体算法略。

7.16 解：

划分算法 1：交替扫描，从后向前找比基准小的记录，再从前向后找比基准大的记录，两者交换。于是将基准右边的区间 R[p+1]～R[q]分成两部分，左部分≤基准，右部分≥基准；然后将基准和左部分的最后一个交换便得到划分结果。算法如下：

```
int Partition_1(list R,int p,int q) {//对 R[p]~R[q]划分，返回基准位置，双向扫描
  int i,j;
  rectype x,y;
  i=p+1;j=q;x=R[p];                  //x 存基准(无序区第一个记录)
  do {
    while(i<j && R[j].key>x.key) j--;//在右侧找<=x 的元素
    while(i<j && R[i].key<x.key) i++;//在左侧找>=x 的元素
    if(i>=j) break;                  //未发现交换对象
    y=R[j];R[j]=R[i];R[i]=y;i++;j--;
  } while(1);
  if(i==j && R[j].key>x.key) j--;   // i=j 时未检测，调整基准位置
  R[p]=R[j];R[j]=x;
  return j;
}
```

划分算法 3：单向扫描，使扫描过的数据分成小值区和大等值区。设当前小值区末位置为 s（大等值区首位置为 s+1），当前扫描位置为 R[i]。若 R[i]比基准小，则把它与 R[s+1]交换，这时小值区扩大一位，s++；否则大等值区自动扩大一位。算法如下：

```
int Partition_3(list R,int p,int q) {//对 R[p]~R[q]划分，返回基准位置，单向扫描
  int i,s;
  rectype x,y;              //y 辅助交换
  s=p;x=R[p];               //x 存基准(无序区第一个记录)
  for(i=p+1;i<=q;i++) {
    if(R[i].key>=x.key) continue;
    s++;y=R[s];R[s]=R[i];R[i]=y;
  }
  R[p]=R[s];R[s]=x;         //基准移到最终位置
  return s;
}
```

7.17 解：将快速排序的递归算法改成非递归算法，需要引进一个栈，最多不超过 n，如果每次都选较大的部分进栈，处理较短的部分，递归深度可降低到 $O(\log_2 n)$。进一步，并不需要将子序列本身入栈，只要将其边界入栈即可。算法如下：

```
void QuickSort2(list R,int n) {
  int s[maxsize*2];         //maxsize≥log2n+1,栈空间（太大时用动态申请和释放）
  int i,j,k,top;                                              （否则系统内部栈会溢出）
  top=-1;
  s[++top]=1;               //预入栈（子序列的 2 个边界端点）
  s[++top]=n;
  while(top!=-1) {
    j=s[top--];
    i=s[top--];
    while(i<j) {
      k=Partition(R,i,j);
      if(k-i<j-k) {         //后部较大，进栈。分划前部
        s[++top]=k+1;
        s[++top]=j;
        j=k-1;
      }
      else {                //前部较大，进栈。分划后部
        s[++top]=i;
        s[++top]=k-1;
        i=k+1;
      }
    }
  }
}
```

易见，将栈改为队列也可，因为快速排序本身并不要求先对哪个子序列进行划分。

7.18 解：

自顶向下的二路归并排序是一种“分治法”：先将区间分成长度相当的前后两部分 R[1]～R[mid]、R[mid+1]～R[high]，各自排序后得到两个有序区间，再将两者归并。而前后两个有序区间的获得（排序）是用同样的方法：对各自的前后两部分排序，再归并。这个过程是递归的，很容易用递归实现，算法如下：

```
void MergeSort2(list R,list R1,int n,int low,int high) {
//对 R 二路归并排序(递归算法)
  int mid;
  if(low>=high) return;
  mid=(low+high)/2;
  MergeSort2(R,R1,n,low,mid);
  MergeSort2(R,R1,n,mid+1,high);
  Merge2(R,R1,low,mid,high);
}
```

但递归中不能交替使用 R1 和 R 作辅助空间，故 Merge2 归并后的结果仍要在原 R 中，这可在归并前将数据从 R 复制到 R1 中（或归并后将数据从 R1 复制到 R 中）：

```
void Merge2(list R,list R1,int low,int mid,int high) {
//合并 R 的两个子表，结果在 R 中
  int i,j,k;
  for(i=low;i<=high;i++)             //子序列复制到 R1
    R1[i]=R[i];
  i=low;j=mid+1;k=low;
  while(i<=mid && j<=high)
    if(R1[i].key<=R1[j].key) R[k++]=R1[i++];    //取小者复制
    else R[k++]=R1[j++];
  while(i<=mid) R[k++]=R1[i++];      //复制左子序列的剩余记录
  while(j<=high) R[k++]=R1[j++];     //复制右子序列的剩余记录
}
```

该算法尚可改进：在子序列复制时，先将第二个子序列逆置，于是合并时，两个子序列从两端向中间推进，推进端互相成为另一端的“监视哨”，这样就不必检查某个子序列是否先处理完的情况了。算法如下：

```
void Merge2(list R,list R1,int low,int mid,int high) {
//合并 R 的两个子表，结果在 R 中，带监视哨技术
  int i,j,k;
  for(i=low;i<=mid;i++) R1[i]=R[i];//左子序列复制到 R1
  i=mid+1;j=high;
  while(i<=high) R1[j--]=R[i++];     //右子序列逆置复制到 R1
  i=low;j=high;k=low;
  while(i<=j)
    if(R1[i].key<=R1[j].key) R[k++]=R1[i++];
    else R[k++]=R1[j--];
}
```

注意，对非递归的二路归并，由于归并前不需要将数据先复制到 R1（奇趟和偶趟归并交替用 R1 和 R 作辅助空间），不能采用上述“监视哨”技术。

易见递归算法比非递归算法简洁些（不用区分子表个数为奇数、偶数或最后子表较短等情况）。与快速排序相比，递归归并排序在对问题“分解”，即划分子区间时比较容易，但在“组合”，即合并子区间时比较困难。

上述归并排序算法的时间复杂度可递归地表示如下：

$$T(n)=2T(n/2)+n \quad , \quad T(1)=c$$

其中 T(n/2)表示归并长度为 n/2 的子表的时间，n 表示归并两个子表的时间。为简单起见，假设表的长度为 2 的乘方，$n=2^k$，则与推导快速排序最好情况下的比较次数类似，从递归方程式可推出 $T(n)=O(n\log_2 n)$，即时间复杂度与非递归算法相当。

但递归时递归栈引起附加时空开销，加上归并前数据预复制的开销，所以实际执行效率常不及非递归算法。

7.19 解：

（1）将新增加的最后一个关键字与其双亲、双亲的双亲等进行比较，直到根逐步进行调整，算法如下：

```
void adjust(keytype key[],int n) {
  int i,j;
  keytype x;
  x=key[n];
  j=n;
  i=j/2;
  while(i>=1 && x>k[i]) {
    k[j]=k[i];
    j=i;
    i=j/2;
  }
  k[j]=x;
}
```

该算法也可用 k[0]作监视哨，即循环开始前使 k[0]=k[n]，则循环条件为 k[0]>k[i]。

（2）建大根堆的过程就是在已有堆上不断插入，算法如下：

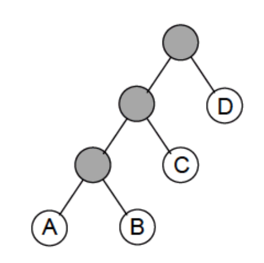
```
void build(keytype key[],int n) {
  int i;
  for(i=1;i<=n;i++) {
    cin>>key[i];
    adjust(key,i);
  }
}
```

（3）当第 i 个结点插入时，二叉树当前结点数为 i，高度为$\lfloor \log_2 i \rfloor + 1$，而插入时最多调整到根，即最多比较次数为$\lfloor \log_2 i \rfloor$，所以总比较次数最多为：

$$
\begin{aligned}
C_1(n) &\leqslant \sum_{i=1}^{n} \lfloor \log_2 i \rfloor \\
&\leqslant \sum_{i=1}^{n} \log_2 i = \log_2 1 + \log_2 2 + \log_2 3 + \cdots + \log_2 n \\
&= \log_2 n! \approx n\log_2 n - 1.44n \leqslant n\log_2 n = O(n\log_2 n)
\end{aligned}
$$

可以证明，平均比较次数约为 2.28n。可见，这种插入式建堆所花费的时间比筛选法建堆要多。

7.20 解：以记录两个数据比较后的较大者为例(若两者相等，任取一个为较大)，可取较大者为根，两个比较的数据为其孩子。于是，所有的比较过程就对应一棵二叉树，显然它不一定是完全二叉树。如右图表示的比较过程为：将 A、B 较大的与 C 比较，取其较大者与 D 比较。这显然不是完全二叉树。



但对锦标赛比较过程，则需用完全二叉树描述。

7.21* 解：由于工作区可容纳 4 个记录，故采用 4 路归并的选择树方法生成初始归并段，其过程和结果见下表所示（选择树略），共生成 3 个初始归并段。

步骤	1	2	3	4	5	6	7	8	9	10	11
工作区内容	16	18	(10)	(10)	(10)	(10)	(10)				
	20	20	20	(13)	(13)	(13)	(13)	(13)			
	43	43	43	43	43	(9)	[5]	[5]	[5]	[5]	
	25	25	25	25	(17)	(17)	(17)	(17)	(17)		
输出结果	[16	18	20	25	43]	[9	10	13	17]	[5]	

7.22* 解：2 路平衡归并排序需要 4 台磁带机，其中，初始归并段应在两条带上交替分布，归并中生成的新归并段应在另两条带上交替分布。归并过程如下（括号表示当前归并段的长度）：

初始：	T1：R1(1L), R3(1L), R5(1L), R7(1L) T2：R2(1L), R4(1L), R6(1L) T3： T4：	第二趟归并：	T1：R1(4L) T2：R2(3L) T3： T4：
第一趟归并：	T1: T2: T3：R1(2L), R3(2L) T4：R2(2L), R4(L)	第三趟归并：	T1: T2: T3：R1(7L) T4:

7.23* 解：2 路多步归并排序需要 3 台磁带机，归并过程如右表所示。可见最后还有 5 个归并段，未能全部归并完（需要将它们重新分配到其他两条带上再进行 2 路归并）。这是因为初始归并段数分布不好。

对本题，从 2 阶 Fibonacci 数“0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …”看，由于 $2f_8+f_7=2\times21+13=55$ 最接近 $15+35=50$，故理想的段数分布为 $T_1=f_8+f_7=21+13=34$，$T_2=f_8=21$。这时需要补充 5 个长度为 0 的空段。

	T1	T2	T3
初始：	15	35	
第一趟：		20	15
第二趟：	15	5	
第三趟：	10		5
第四趟：	5	5	
第五趟：			5

第 8 章　查找表

8.1 解：不是，平衡二叉树才是。

8.2 解：不能要求所有结点的平衡因子都是 0，如只有 4 个结点的二叉树就做不到。可以要求结点的平衡因子尽可能为 0，理想情况就是完全二叉树，但处理起来比较困难。

8.3 解：当二叉树的叶子全部集中在最后的两层（如完全二叉树、二分查找的判定树）时高度最小，当每层只有一个结点（如左单枝、右单枝二叉树）时高度最大，这时平均查找长度最大，为$(n+1)/2$。

8.4 解：深度为 k 的 AVL 树结点数最多时为满二叉树，此时结点总数为 2^k-1。最少结点数可以递推得到 $n_k=n_{k-1}+n_{k-2}+1$，其中 $n_0=0$、$n_1=1$，所以 $n_2=2$、$n_3=4$、$n_4=7$。

一般通式为 $n_h=F_{h+2}-1=\frac{1}{\sqrt{5}}(\phi^{n+2}-\psi^{n+2})-1$，其中 $\phi=\frac{1+\sqrt{5}}{2}$，$\psi=\frac{1-\sqrt{5}}{2}$（见附录 E）

8.5 解：不能，因为散列方法不能表示线性等逻辑关系，只适合表示集合。

8.6 略。

8.7 解：设原散列表为空，则第一个关键字探查 1 次、第二个关键字探查 2 次、……，即最少探查次数为 1+2+3+…+n=n(n+1)/2。

8.8 解：二分法查找过程如下：

```
        序号：  1   2  3  4  5  6   7   8  9  10 11 12 13
第一次比较：[ 3,  5,10,12,17,20,23,27,31,34,39,40,41]
第二次比较：[ 3,  5,10,12,17,20]23,27,31,34,39,40,41]
第三次比较：[ 3,  5,10[12,17,20]23,27,31,34,39,40,41]
第四次比较：[ 3,  5,10[12]17,20,23,27,31,34,39,40,41]
```

查找成功，经过了 4 次关键字比较。

8.9 解：

（1）n=10，高度 h=⌈$\log_2(n+1)$⌉=4，这即最大查找长度。查找成功时，

$$ASL=\frac{n+1}{n}h-\frac{2^h-1}{n}=\frac{10+1}{10}4-\frac{2^4-1}{10}=2.9$$

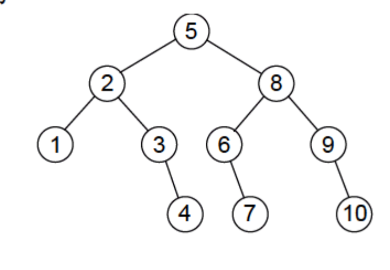
近似计算：ASL≈$\log_2(n+1)-1$=2.46，差别大（因这时 n 较小），或 ASL≈$\frac{n+1}{n}\log_2(n+1)-1$=2.81，差别小。

（2）n=100，高度 h=⌈$\log_2(n+1)$⌉=7，这即最大查找长度。查找成功时，

$$ASL=\frac{n+1}{n}h-\frac{2^h-1}{n}=\frac{100+1}{100}7-\frac{2^7-1}{100}=5.8$$

近似计算：ASL≈$\log_2(n+1)-1$=5.66，差别小，或 ASL≈$\frac{n+1}{n}\log_2(n+1)-1$=5.72，差别更小。

另外，对（1），由于结点数较小，也可先画出二分查找的判定树，见右图所示。查找成功时平均查找长度为：



ASL=(1×1+2×2+3×4+4×3)/10=2.9

对（2），因结点数较大，画出判定树后再分析则不合适。

8.10* 解：对一个具体的二叉排序树而言，其 ASL 为：

$$ASL=\frac{1}{n}\sum_{i=1}^{n}h_i=\frac{1}{n}\sum_{i=1}^{n}[(h_i-1)+1]=\frac{1}{n}\sum_{i=1}^{n}d_i+1$$

其中，h_i 为结点的高度，d_i 为结点的路径长度。$\sum_{i=1}^{n}d_i$ 称为结点的内路径长度。于是，问题转化为求所有 n 结点二叉排序树的平均内路径长度，设用 I(n)表示。显然 I(1)=0，I(2)=1。

对 n 个关键字，有 n!种排列，对应 n!棵二叉排序树（其中有的形态相同）。把这些序列分为 n 种：序列中分别有 0、1、…、n−1 个关键字小于第一个关键字（相应有 n−1、n−2、…、0 个关键字大于第一个关键字）。于是对应二叉排序树的左子树中分别有 0、1、…、n−1 个结点（相应地右子树有 n−1、n−2、…、0 个结点）。等概率时把所有这些情况平均得：

$$I(n)=\frac{1}{n}\sum_{i=0}^{n-1}[I(i)+I(n-i-1)+n-1]$$

$$=\frac{2}{n}\sum_{i=0}^{n-1}I(i)+n-1 \quad (n\geqslant 1)，其中 I_0=0, I_1=0, I_2=1, \cdots$$

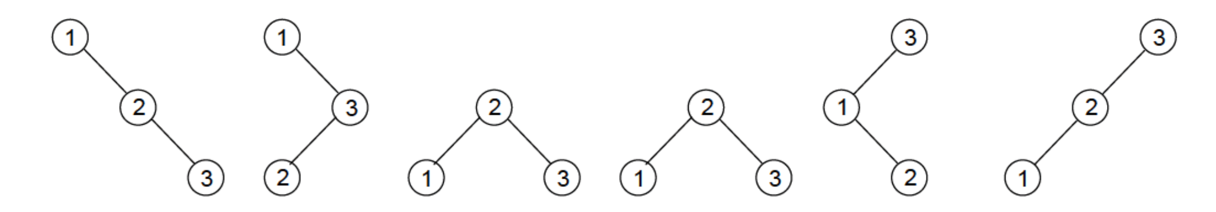
该式与习题 7.10 完全相同，于是

$$I(n)=2(n+1)\ln(n+1)+O(n)$$

所以

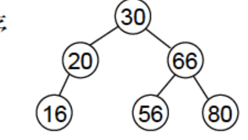
$$ASL=\frac{I(n)}{n}+1=2\frac{n+1}{n}\ln(n+1)+O(1)\approx 2\ln(n)\approx 1.39\log_2 n$$

8.11 解：（1）二叉排序树的形态由输入序列决定，一般 n 个关键字有 n!种输入序列，所以可有 n!个二叉排序树，但其中有的形态相同。对本题，关键字有 3 个，故有 3!=6 种输入序列：1, 2, 3、1, 3, 2、2, 1, 3、2, 3, 1、3, 1, 2、3, 2, 1，相应的二叉排序树如下图所示，其中与序列 2, 1, 3、2, 3, 1 对应的二叉排序树相同，故实际上只有 5 种二叉排序树。



（2）相当于求 n 个结点的二叉树的形态数（每种形态都有自己的中序序列，使其对应到 1, 2, ⋯, n，便得到中序序列递增的二叉树，即二叉排序树），利用习题 5.5（2）的结果便知该数为 $b_n=\frac{1}{n+1}\cdot\frac{(2n)!}{n!n!}=\frac{1}{n+1}C_{2n}^{n}$，即 1, 2, 5, 14, 42, ⋯。

8.12 解：二叉排序树的特点是中序序列为递增有序的，本题中序序列为{16, 20, 30, 56, 66, 80}，将它们对应到图中，结果如右图所示。

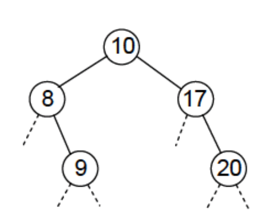


8.13 解：二叉排序树见右图，查找成功时平均查找长度为

$$ASL=\sum_{i=1}^{5}p_i c_i=(1+2\times2+2\times3)/5=2.2$$

查找不成功时有右图中虚线所示的几种情况，平均查找长度为

$$ASL_{un}=\sum_{i=1}^{6}p_i c_i=(2\times2+4\times3)/6=2.67$$



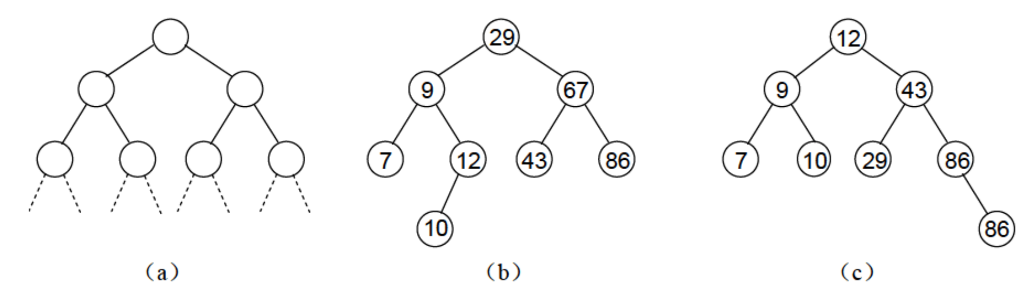
注意这里指关键字比较，不包含空树的空指针比较。

8.14 解：平均查找长度 $ASL=\frac{1}{n}\sum_{i=1}^{n}h_i$，其中 h_i 为结点的深度。为使 ASL 最小，就要使结点深度和最小，这就要使结点尽量填满二叉树的上层，其结果是，最后一层可以不满，其上各层为满二叉树。AVL 平衡调整算法不一定得到这样的结果，如下题的图（h）就不是（输入顺序不同时）。

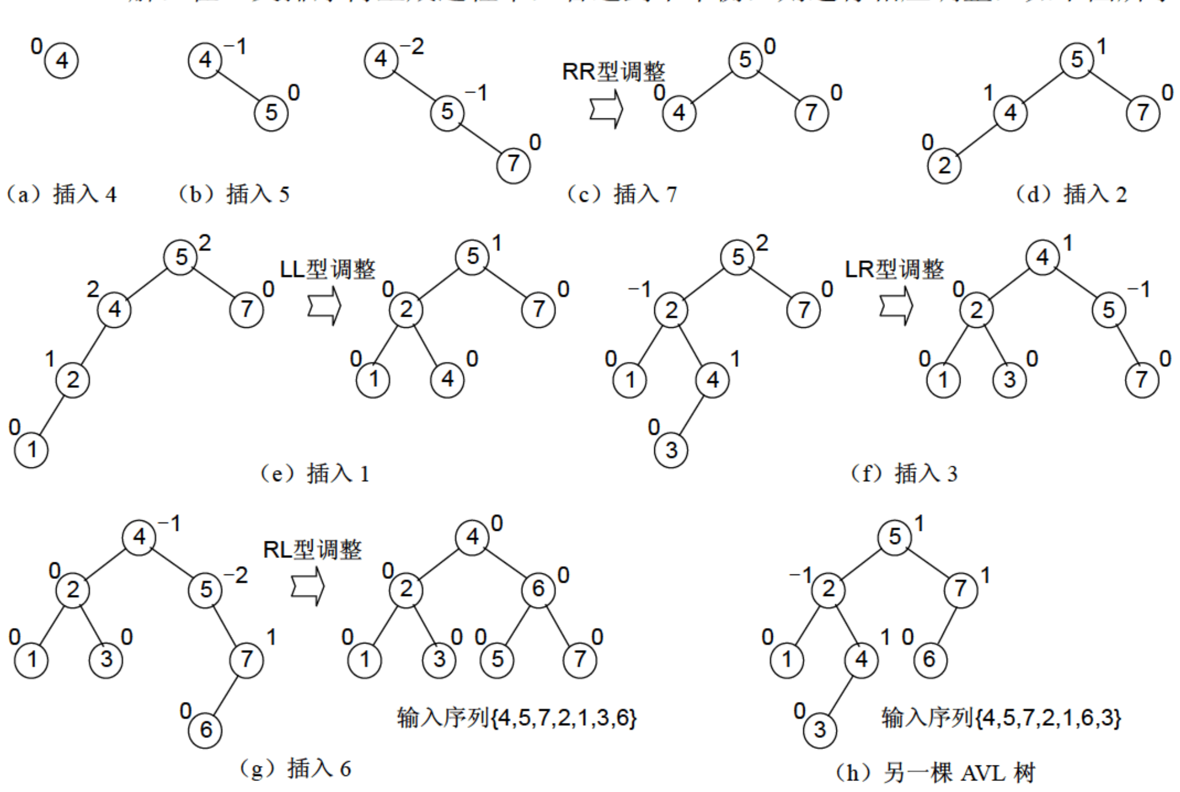
这里给出两个构造方法。其一，根据给定的结点数 n，画出一个满足形态要求的二叉树（若结点数不能正好构成满二叉树，则符合条件的二叉树有多个），然后将其中序序列列出，再将所给关键字排序，按递增顺序对应到二叉树中即可。这就是前面题 8.12 的方法。

其二，将关键字排序，画出对应的二分查找判定树，将各结点填上对应的关键字即可。

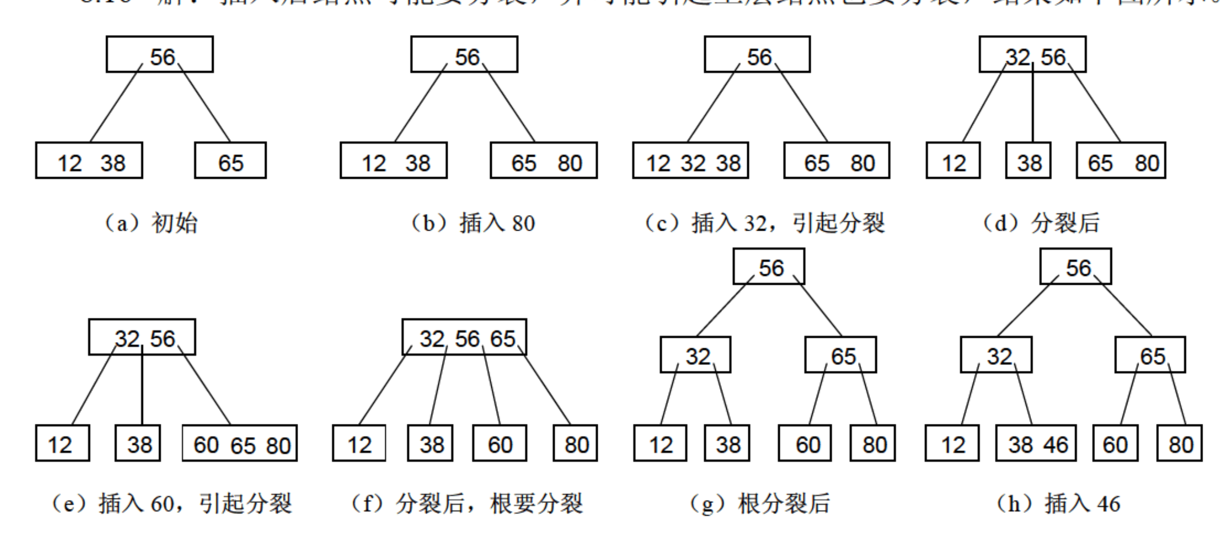
如对关键字集合{7, 29, 10, 9, 43, 67, 12, 86}，n=8，按方法一可以得到 8 棵合乎要求的二叉树，其形态见下图(a)所示，(b)给出了其中的 1 棵；按方法二只能得到 1 棵，见下图(c)。



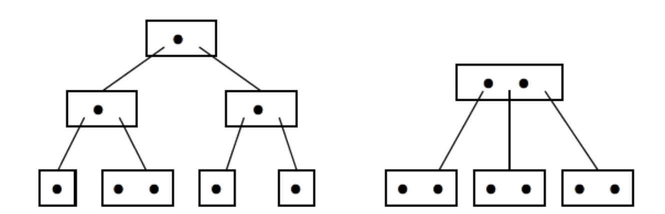
8.15* 解：在二叉排序树生成过程中，若遇到不平衡，则进行相应调整，如下图所示。



8.16* 解：插入后结点可能要分裂，并可能引起上层结点也要分裂，结果如下图所示。



8.17[*] 解：见下图所示，最多 7 个结点，最少 4 个结点，其中“•”表示一个关键字。



8.18[*] 解：这里散列表长度 m=15，故取散列函数为 K%13。各关键字的散列地址见下图（a），按二次探查 $H(K)=(d+i^2)\%m$ 构造的散列表见下图（b）。

（a）

关键字 K	23	15	7	47	28	14	26	53	32	69	81	77
散列地址 K%13	10	2	7	8	2	1	0	1	6	4	3	12

（b）

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
散列表	26	14	15	28	69	53	32	7	47		23		81	77	
成功比较次数	1	1	1	2	1	3	1	1	1		1		4	2	

查找成功时的平均比较次数为：

$$ASL=(1+1+1+2+1+3+1+1+1+1+4+2)/12=1.583$$

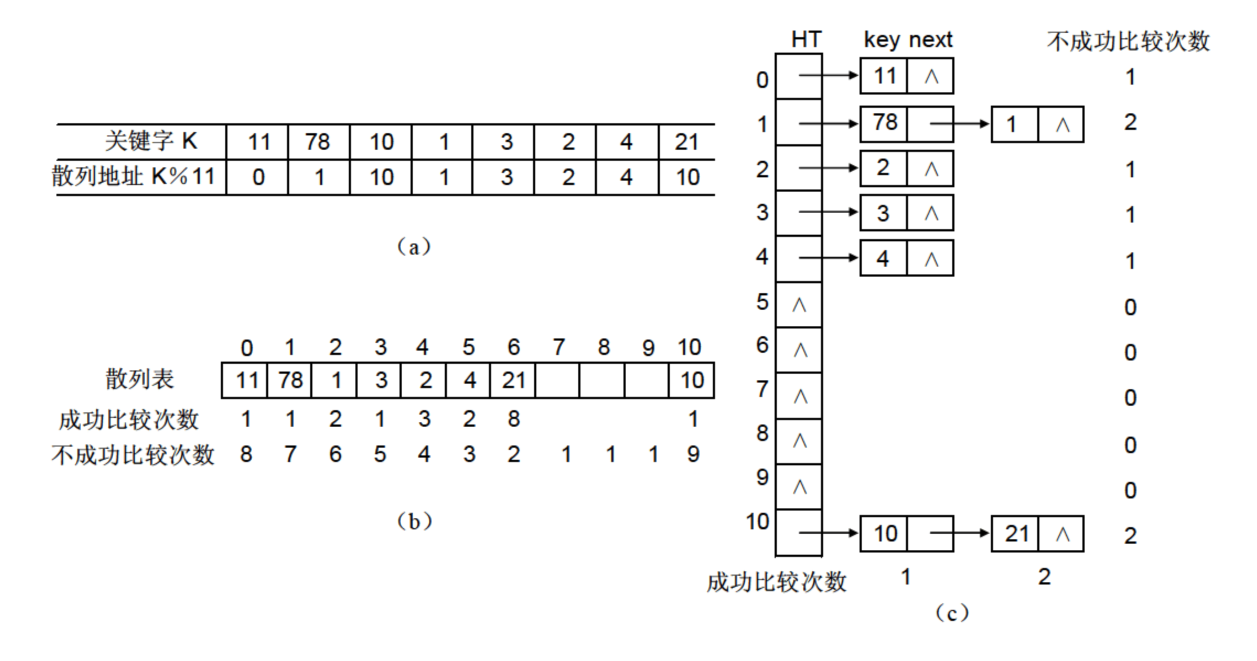
此题装填因子 $\alpha=12/15=0.8$，若用公式计算，则 $ASL\approx-\dfrac{1}{\alpha}\ln(1-\alpha)=2.012$，差别较大。

另外，本题若用 $H(K)=(d\pm i^2)\%m$ 双向二次探查，结果并无优势（略）。

8.19[*] 解：这里没有给出具体数据，只能进行估计。二次探查法查找不成功时的平均查找长度 $ASL_{un}=1/(1-\alpha)\leqslant1.5$，所以 $\alpha\leqslant1/3$，即 $\alpha=n/m=200/m\leqslant1/3$，于是 $m\geqslant600$。二次探查要求表长 m 为满足 4j+3 的质数，故 m 可以取 601。

由于表长 m 为质数，除余法的除数 p 可直接取为 m，所以散列函数 H(K)=K%m。

8.20 解：首先求出散列地址，见下图（a），据此得到闭散列表和开散列表分别见下图（b）和（c）。



关键字 K	11	78	10	1	3	2	4	21
散列地址 K%11	0	1	10	1	3	2	4	10

（a）

	0	1	2	3	4	5	6	7	8	9	10
散列表	11	78	1	3	2	4	21				10
成功比较次数	1	1	2	1	3	2	8				1
不成功比较次数	8	7	6	5	4	3	2	1	1	1	9

（b）

（c）

对线性探查法构造的闭散列表，查找成功的平均比较次数为：

ASL=(1+1+2+1+3+2+8+1)/8=2.375

查找不成功的平均查找长度为：

ASL_{unsucc}=(8+7+6+5+4+3+2+1+1+1+9)/11=4.273

闭散列表装填因子 α=8/11=0.727，若用公式计算，则：

$ASL\approx\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)=2.333$，$ASL_{unsucc}=\frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right]=7.222$

对拉链法构造的开散列表，查找成功的平均比较次数为：

ASL=(1×6+2×2)/8=1.25

查找不成功的平均查找长度为：

ASL_{unsucc}=(1+2+1+1+1+0+0+0+0+0+2)/11=0.727

开散列表装填因子 α=8/11=0.727，若用公式计算，则：

$ASL\approx1+\frac{\alpha}{2}=1.364$，$ASL_{unsucc}=\alpha=0.727$（相同）

8.21 解：即第 2 章顺序表采用的定位（按值查找）算法（略）。

当采用监视哨技术时，将监视哨设在顺序表的末端，即 R[n+1]处，算法如下：

```
int search(sqtable *R,keytype K) {
  int n,i;
  n=R->n;
  R->data[n+1].key=K;            //设置监视哨
  i=1;
  while(R->data[i].key!=K) i++;
  if(i==n+1) return 0;           //查找成功时返回 K 在表中的序号，否则返回 0
  else return i;
}
```

该算法在返回时也可用“return i%(n+1);”来统一处理查找成功和不成功两种情况。

显然，查找成功时找到第 i 个元素所需的比较次数 c_i=i；查找不成功时，while 循环终止于 R->data[n+1].key，算法的时间性能与从后向前进行查找时相同。

8.22 解：即第 2 章链表采用的定位（按值查找）算法（略）。

8.23 解：在递归中要指出查找区间的首末位置，算法如下：

```
int BiSearch2(sqtable *R,int low,int high,keytype K) {
//升序折半查找,递归算法
  int mid;
  if(low>high) return 0;                    //查不到
  mid=(low+high)/2;
  if(K==R->data[mid].key) return mid;
  if(K<R->data[mid].key)  return BiSearch2(R,low,mid-1,K); //在前半部分查找
  else return BiSearch2(R,mid+1,high,K);    //在后半部分查找
}
```

8.24 解：按“右–根–左”的次序进行遍历即可。算法略。

8.25 解：可检查中序序列是否为递增有序。遍历中用 pre 保存结点前趋的值，最后结果由全局量 flag 表示。算法如下：

```
int flag=1;
```

```
keytype pre=KEY_MIN;                 //KEY_MIN 为关键字类型的最小值
void detect(bitree t) {
  if(t==NULL || flag==0) return;    //空树或已为假
  detect(t->lchild);
  if(t->data<=pre) {flag=0;return;}//当前<=前趋，假
  pre=t->data;
  detect(t->rchild);
}
```

如果不想将 flag 和 pre 设成全局量，则可将它们设成函数的参数。但它们应该是双向传递，即按地址传递（或用 C++的引用方式传递）。

8.26 解：

（1）插入过程是从根结点开始逐层向下寻找插入位置的，算法如下：

```
bitree insert2(bitree t,keytype K) {   //非递归算法
  pointer f,p,s;
  p=t;
  while(p!=NULL) {
    if(K==p->key) return t;        //树中已有结点*s，无须插入
    f=p;                           //保存当前结点，它是下一个结点的双亲
    if(K<p->key) p=p->lchild;      //在左子树上找插入位置
    else p=p->rchild;              //在右子树上找插入位置
  }
  s=new node;                      //生成新结点
  s->lchild=NULL;
  s->rchild=NULL;
  s->key=K;
  if(t==NULL) return s;            //原树为空，新结点*s 作为根指针
  if(K<f->key) f->lchild=s;        //将*s 插入为*f 的左孩子
  else f->rchild=s;                //将*s 插入为*f 的右孩子
  return t;
}
```

（2）查找中每次与根比较，由此决定下一步是向左子树还是右子树进行查找，算法如下：

```
pointer Search2(Bitree t,keytype K) {   //非递归算法
  pointer p;
  p=t;
  while(p!=NULL) {
    if(K==p->key) return p;        //查找成功
    if(K<p->key) p=p->lchild;      //在左子树中查找
    else p=p->rchild;              //在右子树中查找
  }
  return NULL;                     //查找失败
}
```

8.27 解：依次对每一个散列地址 1～26，线性探查输出其同义词即可。由于装填因子小于 1，肯定有空单元，故探查中不必检查是否探查了 m 次。算法如下：

```
void disp(hashtable HT) {
  int i,j;
  for(i=1;i<=26;i++) {
    j=i;
    while(HT[j].key!=OPEN) {
      if(HT[j].key==i) cout<<HT[j].key;
```

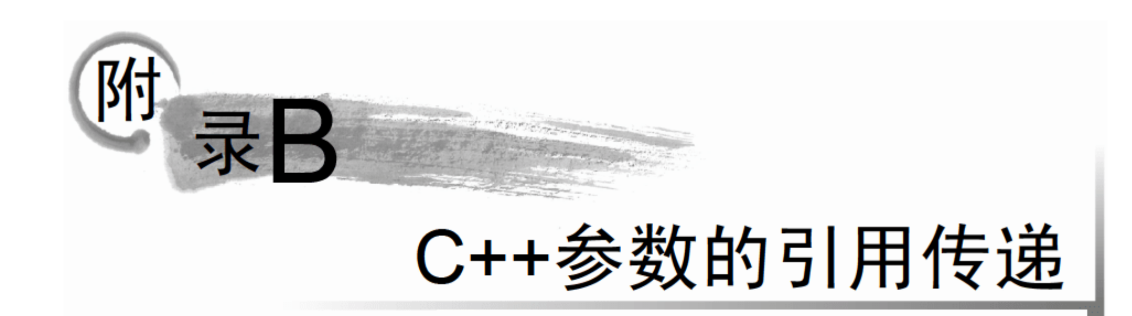
```
      j=(j+1)%m;
    }
  }
}
```

8.28 解：统计每个散列地址下不成功比较次数，再除以表长即可。算法如下：

```
float unsecc(chainhash HT) {
  pointer p;
  float sum;
  sum=0;
  for(i=0;i<m;i++) {
    p=HT[i];
    while(p!=NULL) {sum++;p=p->next;}
  }
  return sum/m;
}
```

第 9 章　文件

答案略。



1. 引用

C++不仅对 C 的已有内容作了很多改进和增强，而且还增加了一些新的功能，引用就是其中之一。简单地说，**引用**就是另一变量（或常量）的别名，对它的使用就是对原变量（或常量）的使用。语法符号为&(不是取地址)。如

```
void main() {
  int  a=5,b=3,c,&d=c;//d 为 c 的引用（或者说 d 为 c 的别名）
  d=a+b;               //修改 d 就是修改 c（结果 c 为 8）
  cout<<c<<endl;
}
```

2. 引用传递

在函数调用时，主调函数和被调用函数间信息的传递一般有三种方式：全局量、函数参数和函数返回值。对参数传递方式，一般又有两种：传值方式和传指针方式。后者本质上也是传值（传指针的值）。传值方式是单向的：被调用函数修改形参时，实参并不改变。调用执行时先将实参内容拷贝到形参，若实参所占空间较大（如结构体），则比较费时。传指针方式是双向的：被调用函数修改形参指针所指向的量后，实参指针所指向的量同时修改（但实参指针本身的值并不变）。

若采用引用参数，则由于形参和实参实质上是同一个量，参数传递时就避免了传值方式（单向传递）时实参到形参的数据拷贝（从而节省时间），又比传指针方式（双向传递）简洁自然（有关量通过参数名直接访问，而不是通过指针间接访问）。以 2 个数的交换和求和为例，这些传递方式的对比如表 B.1 所示。

表 B.1　参数传递方式对比

C 语句	C++语句

int sum(int x,int y) {//传值
return x+y;　　//x,y 为实参 a,b 的拷贝
}
	int sum(int &x,int &y) {//传引用
	return x+y;　　　//x,y 就是实参 a,b 本身
	}

void swap(int *x,int *y) {//传指针 int z; z=*x;*x=*y;*y=z;　//通过指针间接访问 }　　　　　　//x,y 为实参 a,b 的地址	void swap(int &x,int &y) {//传引用 int z; z=x;x=y;y=z;　　　　//通过名字直接访问 }　　　　　　　//x,y 就是实参 a,b 本身

void main() {
int a=5,b=3,c;
c=sum(a,b);　　　//传 a,b 的值
swap(&a,&b);　　　//传 a,b 的地址
…
}
	void main() {
	int a=5,b=3,c;
	c=sum(a,b);　　　　//传 a,b 的引用
	swap(a,b);　　　　//传 a,b 的引用
	…
	}

特别地，对传指针方式，若指针本身的值（不是指针所指对象）需要返回，则参数应为该指针的指针（二级指针），使用更不方便。对此一个解决方法是采用函数值返回修改后的指针。但采用引用传递更简便。以 8.3.1 节二叉排序树的删除为例，并假设待删点 P 有双亲，且为双亲 F 的左孩子 f->lchild，三种方法对比如下。

（1）用函数值返回修改后的指针。

```
pointer del_node(pointer p) {   //p 指向待删点
   pointer q,b;
   if(p==NULL) return;          //空结点，不存在
   if(p->rchild==NULL) {        //P 只有左子树
      q=p; p=p->lchild; delete q;
   }
   else if(p->lchild==NULL) {…}//P 只有右子树
   else {…}                     //P 有 2 个子树
   return p;                    //返回删除后的子树(根)
}
```

该函数的调用形式为 f->lchild=del_node(f->lchild)。

（2）用参数返回修改后的指针。这时参数为指针的地址（即指针的指针，二级指针）。

```
void del_node(pointer *R) {     //R 为指向待删点的指针的指针
   pointer q,b;
   if((*R)==NULL) return;       //空结点，不存在
   if((*R)->rchild==NULL) {     //(*R)只有左子树
      q=(*R);(*R)=(*R)->lchild; delete q;
   }
   else if((*R)->lchild==NULL) {…}//(*R)只有右子树
   else {…}                     //(*R)有 2 个子树
}
```

该函数的调用形式为 del_node(&(f->lchild))。

（3）用引用返回修改后的指针

```
void del_node(pointer &p) {     //p 为指向待删点的指针的引用
   pointer q,b;
   if(p==NULL) return;          //空结点，不存在
   if(p->rchild==NULL) {        //P 只有左子树
      q=p; p=p->lchild; delete q;
   }
   else if(p->lchild==NULL) {…}//P 只有右子树
   else {…}                     //P 有 2 个子树
}
```

该函数的调用形式为 del_node(f->lchild)。

一般地，传值方式都可被传引用方式所取代，但实参所占空间不大且只需单向传递时习惯上还是采用传值方式，如基本类型 char、int、float、double 等。另外，即使采用引用传递方式，也可实现“单向传递”，即在参数表中的该参数前加 const 修饰，以限制被调用函数对该参数的修改，如把前述求和函数写成 int sum(const int &x, const int &y)等。



附录C 排序算法的时间统计

1．绝对时间

在排序算法运行前后，分别用 C/C++的系统函数 clock( )获得当前系统时间，则前后两次的时间差就是绝对时间。但这里的时间单位是系统内部的 TCK 数，把它除以系统每秒钟的 TCK 数便得到以秒为单位的时间。参考用法如下：

```
void main() {
  clock_t t1,t2;
    ⋮
  t1=clock();
  BubbleSort(R,n);
  t2=clock();
  cout<<"时间："<<float(t2-t1)/CLK_TCK<<endl;
}
```

2．逻辑时间

设置计数器 C、M，分别统计关键字的比较和移动次数。为便于多处使用，把计数器设置为全局量。但计数器不宜采用系统提供的标准整型量，因为规模较大时，关键字比较和移动次数很大，这些量会溢出。这里采用内部类型__int64。以直接插入排序为例，参考用法如下：

```
__int64 C,M;                        //比较和移动次数
void InsertSort1(list R,int n) {//直接插入排序，带监视哨
  int i,j;
  for(i=2;i<=n;i++) {               //依次插入 R[2],R[3],…,R[n]
    if(C++,R[i].key>=R[i-1].key) continue;//R[i]位置已正确，本趟不需插入
    M++,R[0]=R[i];                  //R[0]是监视哨
    j=i-1;
    do {                            //查找 R[i]的插入位置
      M++,R[j+1]=R[j];j--;          //记录后移，继续向前搜索
    } while(C++,R[0].key<R[j].key);
    M++,R[j+1]=R[0];                //插入 R[i]
  }
}
```

其中用逗号运算符把计数器插入到有关语句处。注意，在统计绝对时间时，以上计数器语句要去掉。

3．随机数的生成

为了测试排序算法的平均性能，可对均匀分布的随机数序列进行排序。随机数的生成

要用到随机函数。但这里不宜采用系统提供的随机函数 rand( )，因为它生成的随机数范围为 0～RAND_MAX=32 767，当规模大于 RAND_MAX 时，输入的数据序列中必出现大量的重复数据，即序列的“随机性”不好。一种简单做法是取两次随机函数之积作为随机数。下面给出一个常用的随机数生成算法——素数模乘同余法。

```
int random2() {return rand()*rand();} //最大值约为下面 M 的一半
int random3() {          //素数模乘同余法,0~M
  int A=16807;           //或 48271
  int M=2147483647;      //有符号 4 字节最大素数
  int Q=M/A;
  int R=M%A;
  static int x=1;        //种子(设为 1)
  int x1;
  x1=A*(x%Q)-R*(x/Q);
  if(x1>=0) x=x1;
  else      x=x1+M;
  return x;
}
```

为了得到不同的随机数序列，可改变随机数发生器的种子。对系统函数 rand( )要用另一函数 srand( )来设置种子。但要注意，若种子取为系统时间，虽然可获得不同的输入序列，但因每次运行时系统时间不同，于是即使同一排序算法，每次运行的结果一般就不同，即结果难以重复，不便在不同的时间、不同的计算机上进行比较（或者说在比较时数据会有一定的出入）。

由于对不同随机数序列的排序时间有差别，还可以对多个序列的运行结果进行平均以更好地反映算法的平均情况。但一般说来，规模越大，且输入序列的随机性越好，则单次运行的结果就越接近平均结果。



1．单链表综合实验

尽可能多地实现单链表的有关运算，并综合在一起：单链表的建立（头插法、尾插法）、特定结点数统计（长度、正/负值结点数等）；查找（按值、按序号）；删除（按值、按序号）；变换（逆置、正/负值结点分置前后两端等）；性质判断（如是否构成等差、等比、全为正数等）。

2．二叉链表综合实验

尽可能多地实现二叉链表的有关运算，并综合在一起：二叉链表的建立（根据补充虚结点的层次序列、补充虚结点的先序序列、先序加中序序列、后序加中序序列等）；特定结点数统计（结点总数、叶结点数、度 1 结点数、度 2 结点数、正/负值结点数等）；求高度；求宽度；遍历（递归和非递归的先根、中根、后跟遍历；层次遍历等）；性质判断（如是否为完全二叉树、大/小根堆、全为正数等）；特定关系结点输出（如兄弟、双亲、祖先等）。

3．有向图综合实验

尽可能多地实现有向图的有关运算，并综合在一起：图的建立（根据输入的边信息建立邻接矩阵或邻接表）；输出指定顶点信息（出、入度，顶点值，邻接点等）；遍历（递归和非递归的 DFS 遍历、BFS 遍历）；删除、插入顶点；删除、插入边；简单路径问题（指定两点间是否有简单路径、是否有包含所有顶点的简单路径）；简单回路问题（是否有从指定点出发的简单回路、是否有包含所有顶点的简单回路）；求特定距离顶点（距指定点指定最短路径长度的顶点、距指定点最短路径长度最大的顶点等）。

4．排序算法综合实验

实现基于比较的各种基本排序方法，并尽可能给出改进；对不同规模、不同数据集（随机序列、递增序列、递减序列）进行排序，测试算法的绝对时间和逻辑时间（比较次数、移动次数），将结果汇总成表，并与理论结果比较。

5．散列表综合实验

实现开散列表、闭散列表的建立和查找，改变散列函数、冲突处理方法、装填因子等，统计查找成功和不成功时的平均查找长度，将结果汇总成表，并与理论结果比较。

以上几个实验是基础性的，目的是通过上机来体验和掌握课本的有关基本知识，比如二叉链表的实验可考查递归技术的灵活应用，对非递归遍历和层次遍历，可考查栈和队列的使用等。可根据实验学时的多少选做；根据完成情况，还可把内容具体化，如设单链表的结点表示学生信息，则把问题转化为对学生信息进行管理；又如设二叉链表的结点表示家族成员信息，则把问题转化为对族谱进行管理等。

根据学生情况，也可安排一些较单一、偏应用的实验，如哈夫曼编码（最优二叉树的应用）、中缀表达式的转换与求值（栈的应用）等。

附录E 几个数学公式

1．级数和

（1）$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$

（2）$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$

（3）$\sum_{i=0}^{n} 2^i = 1 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$，更一般 $\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{a^{n+1}-1}{a-1}$

（4）$\sum_{i=1}^{k} i \cdot 2^{i-1} = 1 \cdot 2^0 + 2 \cdot 2^1 + 3 \cdot 2^2 + \cdots + k \cdot 2^{k-1} = (k-1) \cdot 2^k + 1$

证：记 $S = 1 \cdot 2^0 + 2 \cdot 2^1 + 3 \cdot 2^2 + \cdots + k \cdot 2^{k-1}$，两边乘以 2，再相减：

$$2 \cdot S = 1 \cdot 2^1 + 2 \cdot 2^2 + 3 \cdot 2^3 + \cdots + k \cdot 2^k$$

$$2 \cdot S - S = S = k \cdot 2^k - (1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + \cdots + 1 \cdot 2^{k-1})$$

$$= k \cdot 2^k - (2^k - 1) = (k-1) \cdot 2^k + 1$$

（5）$\sum_{i=1}^{k} (i-1) \cdot 2^{i-1} = 0 \cdot 2^0 + 1 \cdot 2^1 + 2 \cdot 2^2 + \cdots + (k-1) \cdot 2^{k-1} = (k-2) \cdot 2^k + 2$

2．调和级数

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

近似计算：因为 $H_n = \sum_{i=1}^{n} \frac{1}{i} \Delta x \quad (\Delta x = 1)$

所以 $H_n \approx \int_{1/2}^{n+1/2} \frac{1}{x} dx = \ln\left(n + \frac{1}{2}\right) - \ln\frac{1}{2} \approx \ln(n) + 0.7$

准确计算：$H_n = \ln(n) + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \varepsilon$

其中 $\gamma = 0.577215665\cdots$ 为欧拉常数，$0 < \varepsilon < \frac{1}{252n^6}$

范围估计：$\ln(n) < H_n < 1 + \ln(n)$

3．斯特林（Stirling）阶乘公式

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left[1 + \frac{1}{12n} + o\left(\frac{1}{n^2}\right)\right]$$

取 2 为底的对数：

$$\log_2 n! \approx \left(n+\frac{1}{2}\right)\log_2 n-(\log_2 e)n+\frac{1}{2}\log_2(2\pi)+\frac{\log_2 e}{12n}+o\left(\frac{1}{n^2}\right)$$

粗略估计：$\log_2 n! \approx n\log_2 n-1.44n$ （$\log_2 e \approx 1.4426951$）或 $\log_2 n! \approx n\log_2 n-1.5n$

近似计算：因为 $\ln(n!)=\ln(n)+\ln(n-1)+\cdots+\ln(1)=\sum_{i=1}^{n}\ln(i)\Delta x \quad (\Delta x=1)$

所以 $\ln(n!)\approx\int_{1/2}^{n+1/2}\ln(x)dx=(x\ln x-x)\Big|_{1/2}^{n+1/2}=\left(n+\frac{1}{2}\right)\ln\left(n+\frac{1}{2}\right)-n+\frac{1}{2}\ln 2$

准确计算：$\ln(n!)\approx\left(n+\frac{1}{2}\right)\ln(n)-n+\frac{1}{2}\ln(2\pi)+\frac{1}{12n}+o\left(\frac{1}{n^2}\right)$（即斯特林公式取自然对数）

4．斐波那契数列（Fibonacci Sequence）的通项

$F_0=0,\ F_1=1,\ F_n=F_{n-1}+F_{n-2}\ (n\geqslant 2)$

取斐波那契数为系数构造多项式（生成函数）：

$F(x)=F_0+F_1x+F_2x^2+\cdots+F_nx^n+\cdots$

两边分别乘 x、x^2，再相减：

$xF(x)=F_0x+F_1x^2+\cdots+F_{n-1}x^n+\cdots$

$x^2F(x)=F_0x^2+\cdots+F_{n-2}x^n+\cdots$

$(1-x-x^2)F(x)=F_0+(F_1-F_0)x=x$

所以 $F(x)=\dfrac{x}{1-x-x^2}$

把它拆开：

$F(x)=\dfrac{x}{1-x-x^2}=\dfrac{x}{(1-\phi x)(1-\psi x)}=\dfrac{1}{\sqrt{5}}\left(\dfrac{1}{1-\phi x}-\dfrac{1}{1-\psi x}\right)$，其中 $\phi=\dfrac{1+\sqrt{5}}{2}$，$\psi=\dfrac{1-\sqrt{5}}{2}$。

将 $1/(1-\phi x)$ 和 $1/(1-\psi x)$ 泰勒展开并整理得：

$$F(x)=\frac{1}{\sqrt{5}}\ (1+\phi x+\phi^2x^2+\cdots-1-\psi x-\psi^2x^2-\cdots)$$

与原生成函数对比系数，得：

$$F_n=\frac{1}{\sqrt{5}}\ (\phi^n-\psi^n)$$

由于 $|\psi|\approx 0.618034<1$，故 n 较大时：

$$F_n\approx\frac{1}{\sqrt{5}}\ \phi^n$$

5．卡特兰（Catalan）数列的通项

$b_0=1,\ b_n=\sum_{i=0}^{n-1}b_ib_{n-i-1}\ (n\geqslant 1)$

构造生成函数：

$F(x)\ =b_0+b_1x+b_2x^2+\cdots+b_nx^n+\cdots$

将其平方得：

$$F^2(x)=b_0^2+2b_0b_1x+(2b_0b_2+b_1^2)x^2+(2b_0b_3+2b_1b_2)x^3+(2b_0b_4+2b_1b_3+b_2^2)x^4+\cdots$$
$$=b_0^2+(b_0b_1+b_1b_0)x+(b_0b_2+b_1b_1+b_2b_0)x^2+(b_0b_3+b_1b_2+b_2b_1+b_3b_0)x^3+\cdots$$
$$=b_1+b_2x+b_3x^2+b_4x^3+\cdots$$
$$=[F(x)-1]/x \quad (b_0=1)$$

所以 $xF^2(x)-F(x)+1=0$

但 x=0 时该式退化为线性，二次求根公式不通用。把它改写：

$$[xF(x)]^2-[xF(x)]+x=0$$

则

$$xF(x)=\frac{1\pm\sqrt{1-4x}}{2}$$

由于 x=0 时 xF(x)=0，所以：

$$xF(x)=\frac{1-\sqrt{1-4x}}{2}$$

将该式泰勒展开，与原 xF(x)对比，第 n+1 项的系数即为 b_n。

先对 $(1-4x)^{\frac{1}{2}}$ 泰勒展开，其第 n+1 项为：

$$\frac{\alpha(\alpha-1)(\alpha-2)\cdots(\alpha-n)}{(n+1)!}(-4x)^{n+1} \quad \left(\alpha=\frac{1}{2}\right)$$
$$=-\frac{1\cdot 3\cdot\cdots\cdot(2n-1)}{(n+1)!}2^{n+1}x^{n+1}$$
$$=-\frac{(2n)!}{(n+1)!n!}2x^n$$

所以 $xF(x)=\frac{1}{2}-\frac{1}{2}(1-4x)^{\frac{1}{2}}$ 展开式的第 n+1 项系数，即

$$b_n=-\frac{1}{2}\left[-\frac{(2n)!}{(n+1)!n!}2\right]=\frac{1}{n+1}\cdot\frac{(2n)!}{n!n!}=\frac{1}{n+1}C_{2n}^n$$

6．快速排序平均比较次数、二叉排序树平均内路径长度通项

$C(n)=\frac{2}{n}\sum_{k=0}^{n-1}C(k)+n-1 \quad (n\geqslant 1)$，其中 $C_0=0, C_1=0, C_2=1,\ldots$

两边乘以 n：

$$nC(n)=2\sum_{i=0}^{n-1}C(i)+n^2-n \quad (n\geqslant 1)$$

于是：

$$(n-1)C(n-1)=2\sum_{i=0}^{n-2}C(i)+(n-1)^2-(n-1) \quad (n\geqslant 2)$$

两式相减：

$nC(n)-(n-1)C(n-1)=2C(n-1)+2n-2$，即

$$nC(n)=(n+1)C(n-1)+2n-2 \quad (n\geqslant 2)$$

易见 n=1 也成立。上式两边除以 n(n+1)，注意$\frac{1}{n(n+1)}=\frac{1}{n}-\frac{1}{n+1}$，逐级递推：

$$\frac{C(n)}{n+1}=\frac{C(n-1)}{n}+2\frac{1}{n+1}-2\left(\frac{1}{n}-\frac{1}{n+1}\right)$$

$$\frac{C(n-1)}{n}=\frac{C(n-2)}{n-1}+2\frac{1}{n}-2\left(\frac{1}{n-1}-\frac{1}{n}\right)$$

…

$$\frac{C(1)}{2}=\frac{C(0)}{1}+2\frac{1}{2}-2\left(\frac{1}{1}-\frac{1}{2}\right)$$

全部相加：

$$\begin{aligned}\frac{C(n)}{n+1}&=\frac{C(0)}{1}+2\left(\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n+1}\right)-2\left(1-\frac{1}{n+1}\right)\\&=C(0)+2(H_{n+1}-1)-2\left(1-\frac{1}{n+1}\right)\\&=2H_{n+1}-4+\frac{2}{n+1}\end{aligned}$$

所以 $C(n)=2(n+1)H_{n+1}-4(n+1)+2$

$$\approx 2(n+1)\ln(n+1)+(2\gamma-4)(n+1)+2$$

$$\approx 2n\ln(n)+(2\gamma-4)n$$

其中 $H_n=1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$ 为调和级数，$H_n\approx\ln(n)+\gamma$（见前述结果）。

注意，快速排序有不同的划分算法，其中比较次数可能多于 n−1 次，如 n 次、n+1 次，甚至更多，一般形式为：

$$C(n)=\frac{2}{n}\sum_{k=0}^{n-1}C(k)+\alpha n+\beta$$

这时 C_0、C_1、C_2 等也会不同，类似推导可知，虽然最终表达式会略有不同，但复杂性的数量级相同，都为 $O(\log_2 n)$。

参 考 文 献

[1] 唐策善，李龙澍，黄刘生. 数据结构——用 C 语言描述. 北京：高等教育出版社，1995

[2] 严蔚敏，吴伟民. 数据结构（第二版）. 北京：清华大学出版社，1992

[3] 张晓莉，罗文劼，刘振鹏，许百成. 数据结构与算法. 北京：机械工业出版社，2002

[4] [美]Clifford A.Shaffer 著，张铭、刘晓丹等译. 数据结构与算法分析(C++版)(第二版). 北京：电子工业出版社，2002

[5] [美]Bruno R. Preiss 著，胡广斌，王崧，惠民等译. 数据结构与算法——面向对象的 C++设计模式. 北京：电子工业出版社，2000

[6] 黄刘生. 数据结构. 北京：经济科学出版社，2000

[7] 陈小平. 数据结构导论. 北京：经济科学出版社，2000

[8] 李根强，谢月娥，谢永红，王希辰. 数据结构（C++描述）. 北京：中国水利水电出版社，2001

[9] 崔伟宏.空间数据结构研究. 北京：中国科学技术出版社，1995

[10] 胡元义，邓亚玲，徐睿琳. 数据结构课程辅导与习题解析. 北京：人民邮电出版社，2003

[11] 全国高等教育自学考试命题研究组. 数据结构导论应试指导及模拟试题. 北京：中国大地出版社，2002

[12] 李春葆. 数据结构考研指导. 北京：清华大学出版社，2003

[13] 朱战立，张选平. 数据结构学习指导与典型题解. 西安：西安交通大学出版社，2002

[14] Robert L.Kruse, Clovis L.Tondo, Bruce P.Leung. Data Structures & Program Design in C(2nd ed.)（影印版）. 北京：清华大学出版社，1998

[15] Robert Lafore. Data Structures & Algorithms in Java(2nd ed.)（影印版）. 北京：中国电力出版社，2007